



*Windows Scripting Technology*

# ***RapidBATCH 5 Benutzerhandbuch***

## Inhaltsverzeichnis

Was ist RapidBATCH?.....	3
Was ist neu in Version 5?.....	4
Verbesserungen in Version 5.1 .....	6
Wer entwickelt RapidBATCH?.....	6
Die Grundlagen.....	7
Ein einfaches Script.....	7
Variablen.....	9
Werte manipulieren .....	11
Bedingungen und Schleifen .....	12
Sprünge und Subs.....	13
Arrays .....	15
Prozedurale Programmierung .....	18
Benutzerdefinierte Prozeduren und Funktionen .....	18
Rekursive Programmierung von Prozeduren.....	20
Zeiger auf Variablen .....	22
Build-in Dialogfenster.....	25
Meldungen und Statusausgaben .....	25
Abfragen von Werten.....	28
Menüs und Auswahlmöglichkeiten.....	30
Datei- und Verzeichnisauswahl.....	32
Systemspezifische Operationen .....	36
Programme starten.....	36
Verzeichnisoperationen .....	37
Dateioperationen im Filesystem.....	38
Dateien lesen und schreiben .....	40
Programme fernsteuern.....	44
Bearbeiten der Registry.....	45
Systemspezifische, vordeklarierte Variablen .....	48
Benutzerdefinierte Dialogprogrammierung .....	52
Erstellung von Dialogfenstern .....	52
Die Widgets im Überblick.....	54
Entwicklung einer Beispielanwendung .....	56
Die DLL-Schnittstelle .....	63
Verwendung externer Funktionen .....	63
DLLs selber programmieren .....	63
Ein Beispielprojekt.....	64
Programmablauf und -steuerung .....	72
INCLUDE, INCLUDE_ONCE .....	79
Build-in Funktionen .....	84
Build-in Dialoge.....	89
Systemspezifische Operationen .....	98
Benutzerdefinierte Dialoge .....	116
Debugger Kontrollanweisungen .....	130
Vordeklarierte Variablen.....	132
Index Sprachreferenz und vordefinierte Variablen.....	138

## Was ist RapidBATCH?

RapidBATCH ist eine beliebte Scriptsprache für Windows, deren Ziel die schnelle und unkomplizierte Lösung kleinerer bis mittlerer Software-Probleme ist. RapidBATCH verfolgt dabei eine ganz bestimmte Philosophie: Mit wenig Code viel erreichen!

Bereits die folgend abgebildeten zwei Zeilen stellen ein vollwertiges und ablauffähiges, in RapidBATCH programmiertes Software-Programm dar.

```
echo 'Hallo, ich bin ein RapidBATCH-Script'  
end
```

Zwar sind RapidBATCH-Programme - die so genannten Scripts - von aussen her meist sehr schlicht und einfach gehalten; Unter der "Motorhaube" steckt jedoch mehr als so manch einer erwarten würde. Mit RapidBATCH können fast alle kleineren und größeren Software-Probleme in wenigen Minuten gelöst werden, es ist nur eine Frage der Umsetzung.

RapidBATCH bietet wirklich alles, was man zur schnellen Entwicklung kleinerer Programme (und besonders für erweiterte Batch-Automatisierungsprogramme) benötigt: Dateien erstellen, lesen, schreiben, kopieren, verschieben, löschen und Dateirechte und -daten setzen und verändern, externe Programme starten, Registry bearbeiten, Windows herunterfahren oder sperren, Programme fernsteuern, Mausklicks simulieren, Multimedia-Inhalte abspielen und Dokumente und Internet-URLs aufrufen.

Des weiteren stehen Ihnen zur Interaktion mit dem Benutzer zehn vielseitig einsetzbare Benutzer-Interaktionsdialoge zur Verfügung; Sie können sogar völlig individuelle Dialogfenster mit über 10 verschiedenen Fensterelementen selber programmieren und gestalten.

Als eine moderne prozedurale Programmiersprache bietet Ihnen RapidBATCH auch auf der programmiertechnischen Ebene alles, was Sie zur schnellen Lösung Ihrer Software-Probleme benötigen: Klar strukturierte und flexible Syntax, einfach zu erlernende Befehle und Funktionen, Iterationen (Schleifen), zahlreiche build-in Funktionen zur Verarbeitung von Daten, benutzerdefinierbare Prozeduren und Funktionen, Rekursion (Selbstaufruf von Prozeduren), komplett dynamische Arrays mit bis zu 9 Dimensionen und die Aufteilung von größeren Scripts in mehrere Dateien, welche dann mittels Include-Anweisungen zur Laufzeit virtuell zu einem Script zusammengeführt werden.

RapidBATCH ist für Neulinge extrem simpel zu erlernen, bietet aber auch professionellen Programmierern eine exzellente Basis zur schnellen und unkomplizierten Entwicklung von Softwaretools für den täglichen Gebrauch.

RapidBATCH-Scripts können beispielsweise hingegen zu größeren Programmiersprachen durch einfachen Doppelklick auf die Scriptdatei - wie bei einem EXE- oder Batchprogramm - gestartet werden. Auch ein Projektordner oder etwaige Projekt-Dateien, wie sie von großen Entwicklungsumgebungen bekannt sind entfallen gänzlich.

Die Scripts werden dabei zur Ausführung von dem RapidBATCH-Interpreter interpretiert, d.h. der Script-Code wird im Klartext eingelesen und ausgeführt. Wer in RapidBATCH programmierte Tools weitergeben möchte kann allerdings auch mit Hilfe des RapidBATCH-Compilers eine eigenständig ausführbare EXE-Datei erzeugen, die ohne einen RapidBATCH-Interpreter auf jedem Windows-System ausführbar ist.

Auch bei der Erstellung der Scripts macht es Ihnen RapidBATCH so einfach und angenehm wie nur möglich: Der Script-Editor RapidBATCH Builder ermöglicht die komfortable Erstellung der Scripts, die direkt aus dem Editor heraus gestartet und getestet werden können.

Mit dem integrierten Script-Debugger des RapidBATCH-Interpreters können Sie Ihre Scripts schrittweise Anweisung für Anweisung ausführen und somit deren Ablauf verfolgen; Variableninhalte können zur Laufzeit verändert werden. Somit haben Sie ein nützliches Werkzeug zur Suche von logischen Programmierfehlern zur Hand.

Dieses Handbuch wird Sie nun Schritt-für-Schritt in die Programmierung von Scripts in RapidBATCH 5 einführen. Da RapidBATCH inzwischen zu einer stattlichen und mächtigen Programmiersprache angewachsen ist, behandelt dieses Handbuch aber nur die wichtigsten Elemente und Spracheigenschaften, die RapidBATCH bietet. Des weiteren steht Ihnen jedoch zur detaillierten Ergänzung aller Sprachelemente eine ausführliche und übersichtliche Befehls- und Funktionsreferenz zu allen RapidBATCH-Programmieranweisungen und Funktionen zur Verfügung.

Wir wünschen Ihnen viel Spass und Erfolg beim Programmieren! Danke, dass Sie sich für RapidBATCH entschieden haben.

## Was ist neu in Version 5?

Mit der neuen Version 5 der Scriptsprache RapidBATCH beginnt eine neue Ära in der erfolgreichsten Produktreihe der Firma J.M.K S.F. Software Technologies.

RapidBATCH 5 setzt auf einer komplett neu entwickelten Technologie und dem Know-How von RapidBATCH 4 auf, und wurde innerhalb der fast zweijährigen Entwicklungs- und Testphase komplett überarbeitet und stark erweitert. RapidBATCH ist jetzt quasi nicht mehr nur als Batch- und Automatisierungssprache geeignet, sondern gleichzeitig auch ein leistungsstarkes und effizientes Programmiersystem für kleinere und größere Anwendungen aller Art.

Hier die neuen, kaum zu übertreffenden Features von RapidBATCH 5.0 hingegen zu RapidBATCH 4.4 im Überblick. Beachten Sie bitte, dass einige der neuen Features nicht auf die kostenlose "Personal Edition" von RapidBATCH zutreffen.

- 60% schnellere Script-Ausführung als in RapidBATCH 4.4, dank des neuem Speichermanagements und des überarbeiteten Frameworks.
- 90% Abwärtskompatibel mit RapidBATCH 4.4: RapidBATCH 4.4 Scripts können mit nur minimalster Änderung in einer RapidBATCH 5.0 Scriptumgebung ausgeführt werden.
- Jetzt mit Unterstützung echter Arrays mit bis zu 9 Unterdimensionen. Ein Array kann dynamisch aus jeder RapidBATCH Variable erzeugt werden, Werte können in jede Array-Dimension und in jedem Index hinterlegt werden, auch wenn diese Array-Dimension eine weitere Unterdimension besitzt!
- Voll dynamische, interne Speicherverwaltung: Alle internen Funktionalitäten, die in RapidBATCH 4 noch statisch waren, wurden jetzt komplett dynamisiert.
- Variablen und Arrays können mit der neuen RESET-Anweisung zurückgesetzt (d.h. entleert) werden. Auch das zurücksetzen einzelner unterdimensionen eines Arrays ist möglich.
- Benutzerdefinierte Prozeduren und Funktionen können jetzt in RapidBATCH implementiert und verwendet werden. Dies ermöglicht nun echte, prozedurale Programmlogiken mit verschachtelten Prozeduraufrufen und Rekursionen. Variablen können über das Call-By-Value oder über das Call-by-Reference Verfahren übergeben werden.
- Die sechs RapidBATCH-Operatoren (+, -, \*, /, ~ und #) können jetzt überall in Strings angegeben werden, ausserdem werden jetzt Klammern unterstützt, um Formeln schneller und effizienter definieren zu können.
- Punkt- vor Strichrechnung in allen mathematischen Ausdrücken (ab RapidBATCH 5.1.0).
- Verbesserte Vergleichsoperatoren für Bedingungen: Größer-gleich ( $\geq$ ) und kleiner-gleich ( $\leq$ ) (enthalten ab RapidBATCH v5.0.5).
- Implementierung von "Quick-Assignments": Durch Angabe eines RapidBATCH Standard-Operators anstatt des Gleichheitszeichens bei einer Zuweisung wird dieser Operator mit der zuzuweisenden Variablen verbunden und gleichzeitig zugewiesen. Dies spart viel Tipparbeit bei z.B. Inkrementationen oder beim verbinden von Strings.
- Dynamische Sprünge via GOTO und GOSUB ermöglichen es, einen variablen Label-Namen als Sprungziel anzugeben, zu dem gesprungen wird.
- Virtuelle Scriptzusammenführung: Mit den Anweisungen INCLUDE und INCLUDE\_ONCE können jetzt mehrere RapidBATCH Scriptdateien zur Laufzeit virtuell zu einem großen Script zusammengefügt werden. Dies ermöglicht die Definition von Funktions- und Prozedurbibliotheken, die in unterschiedlichen Scripts angewandt werden können.
- Neuer build-in Dialog TRAYMENU zum Bereitstellen eines Menüs im Windows System-Tray in der Taskleiste.
- Neuer build-in Dialog PWDBOX zum Anzeigen und Bereitstellen eines maskierten Eingabedialogs zur Eingabe von Passwörtern.
- Neue Schleifenkontroll-Statements BREAK und CONT zum Verlassen bzw. Fortführen von REPEAT...UNTIL-Schleifen.
- Erweiterte, zeichenorientierte Dateiverarbeitung mit GETFILE und PUTFILE, womit das lesen und schreiben von Binärdaten ermöglicht wird.
- Mit Hilfe der neuen Funktionalitäten NEWDIALOG, LETDIALOG, GETDIALOG, DELDIALOG und RUNDIALOG können jetzt komplett benutzerdefinierte, individuelle Anwendungsfenster und Dialoge entworfen und programmiert werden. Dieses mächtige Feature macht RapidBATCH damit zu einer vollwertigen Programmiersprache zur Entwicklung von Windows-Anwendungen aller Art! Es können bis zu 17 unterschiedliche Dialog-Elemente (so genannte "Widgets" völlig dynamisch erzeugt und bereitgestellt werden. Jedes Widget unterstützt verschiedene Attribute und Ereignisse, auf die reagiert werden kann.
- Neue Registry-Bearbeitungsfunktionalitäten NEWREGKEY, DELREGKEY, LETREGVAL, GETREGVAL und DELREGVAL.
- Neue Systemfunktionalitäten DISKFREE, DISKSIZE, DISKTYPE, LOCKWORKSTATION und SHUTDOWN.
- Komplett überarbeitete SENDKEYS-Makrosprache mit viel einfacherem Handling als zuvor (ab RapidBATCH 5.1.0).

- Aufruf individueller DLL-Funktionen über die neue EXT-Funktionsschnittstelle: Durch diese Möglichkeit kann so gut wie jede Programmieraufgabe elegant und professionell gelöst werden (ab RapidBATCH 5.1.0)!
- Programm- und Dokumentenverknüpfungen anlegen mit SHORTCUT.
- Fernsteuern der Maus mit MOUSEMOVE, sowie auslesen und setzen der Mausposition mit den neuen, vordeklarierten Variablen [mouse\_x] und [mouse\_y].
- Neue build-in Funktionen CNTVAR, zur Ermittlung der Anzahl eines Substrings innerhalb eines Strings, und GETCHARAT, zur Ermittlung eines Zeichens an einer Position innerhalb eines Strings.
- Die Anweisung MSGBOX kann jetzt wahlweise als Funktion mit Rückgabewert oder als Anweisung aufgerufen werden; Dies wurde Implementiert, da die MSGBOX-Anweisung durch die MSGBOX-Funktion ersetzt werden soll, aber trotzdem eine Kompatibilität zu RapidBATCH 4 bestehen sollte.
- Erweiterte Möglichkeiten für GETTOK, GETPOS, REPLACEVAR und CNTVAR mittels der Variablen [case\_sensitivity], bei denen jetzt auch zwischen Gross-/Kleinschreibung von Substrings unterschieden werden kann.
- Durch setzen der Variablen [std\_sep] kann jetzt der globale Standard-Separator, der ursprünglich durch ein Pipe-Zeichen (|) repräsentiert wurde, geändert und damit auf jedes beliebige Zeichen gesetzt werden. Er gilt damit für alle Anweisungen und Funktionen, die den Standard-Separator verwenden, wie z.B. LISTBOX oder FINDFILES.
- Einführung der Pseudo-Variablen [null] zur Zuweisung bzw. Angabe irrelevanter Informationen, die z.B. von Funktionen zurückgegeben werden oder als nicht benötigte Parameter spezifiziert werden.
- Neue Variablen [true] und [false] als Substitution für '0' und '-1', z.B. in Abfragen ist [a] = [true] aussagekräftiger als [a] = '0'.
- Ermitteln des Titels des jeweils aktiven Vordergrundfensters mit Hilfe der Variablen [active\_window].
- Neue Variable [app\_icon] zum ändern des Fenster- und Programmicons für build-in Dialoge.
- Definition einer Default-Dateierweiterung für die build-in Dialoge OPENBOX und SAVEBOX über die Variablen [OpenBox\_DefFileExt] und [SaveBox\_DefFileExt].
- Neue, vordeklarierte Hilfsvariablen mit Sonderzeichen: [crlf] (carriage return + line feed), [tab] (Tabulator), [pipe] (Pipe-Zeichen) und [quot] (einfaches Anführungszeichen).
- Speziell für Windows XP und Windows 2003: Die Variablen [XPStyleActive], [Metric\_Edge\_Width], [Metric\_Edge\_Height], [Metric\_Caption\_Height] und [Metric\_Menu\_Height] zur Optimierung von benutzerdefinierten Dialogfenstern bei der Verwendung von Windows XP Themes.
- Mitgelieferte, erweiternde RapidBATCH Funktionalitäten, die durch sieben unterschiedliche Bibliotheken bereitgestellt werden (z.B. Array-Bearbeitungsfunktionen, erweiterte Systemfunktionen, erweiterte Dialogfunktionen sowie weitere Universal-Dialoge, u.v.m.). Diese Bibliotheken können per INCLUDE-Anweisung eingebunden werden, und wurden selber zu 100% in RapidBATCH implementiert.
- Verbesserter build-in Script-Debugger mit Prozedurschritt-Funktionalität und verbesserter Verwendbarkeit.
- Komplett neu entwickelter Programmiereditor RapidBATCH Builder mit deutlich schnellerem Syntax-Highlighting und vielen neuen Funktionen, wie z.B. Dateihistorie, benutzerkonfigurierbares Werkzeug-Menü und automatischer Kommentierfunktion.
- Teile des RapidBATCH Software-Pakets wurden sogar in RapidBATCH selbst realisiert. Dazu gehören u.a. die zahlreichen Hilfswerkzeuge, wie z.B. der Visual Dialog Designer, ein Werkzeug zum visuellen, komfortablen erstellen von individuellen Dialogfenstern in RapidBATCH, oder der HTML Script-Code Highlighter, der den nötigen HTML-Code für die farbliche Hervorhebung von RapidBATCH Script-Code generiert.
- Komplett neu geschriebenes, ausführliches Anwenderhandbuch in Form einer Online-Hilfe mit Alphabetischer und nach Sachthemen geordneter Befehls- und Funktionsreferenz sowie farblich hervorgehobenen Programmbeispielen und Listings und Deutschsprachigen Screenshots.
- Komplett Sprachunabhängiges Softwarepaket: Alle RapidBATCH-Entwicklungstools bis auf den Compiler sind jetzt komplett auf Deutsch verfügbar!
- Es kann jetzt jedes beliebige Icon mit einer compilierten RapidBATCH-Scriptdatei gelinkt werden, auch Icons im Windows XP Stil mit 256 Farben und einer Größe von 64x64 Pixel.
- Weiterhin ist RapidBATCH 5 auf allen Windows-Versionen ab Windows 95 lauffähig, wurde aber speziell für den Einsatz auf Windows 2000 und Windows XP optimiert.

Mit diesen vielen Neuerungen wird RapidBATCH jetzt zu einer Plattform zur Entwicklung und Lösung unterschiedlichster Aufgaben und Software-Probleme. Umsteigern von RapidBATCH 4 wird empfohlen, trotz der fast 100%tigen Abwärtskompatibilität zu RapidBATCH 4.4 das komplette, neue Handbuch von Anfang an durchzulesen. Dort sind zahlreiche Tipps und Details zu den neuen Spracherweiterungen von RapidBATCH enthalten, die Sie in Zukunft unbedingt beachten und auch ausnutzen sollten.

## Verbesserungen in Version 5.1

Der aktuelle Release 5.1 kommt mit zahlreichen Neuerungen und kleineren Bug- und Hotfixes, und macht RapidBATCH 5 jetzt noch produktiver und flexibler als je zuvor.

- Punkt- vor Strichrechnung gilt jetzt in allen Ausdrücken: '5' + '2' \* '3' ergibt ab sofort 11 und nicht 21!
- Neues Widget: Mit der Progressbar kann jetzt auch eine Fortschrittsanzeige auf individuellen Dialogen programmiert werden.
- Neues Widget: Das STATIC\_COMBO-Widget zeigt eine statische Combobox an, in der zwar Werte aus einer vorgegebenen Auswahl (Dropdown-Box) ausgewählt werden, jedoch nicht individuell editiert werden können.
- Neu definierte Makrosprache für die SENDKEYS-Anweisung: Anwendungen können jetzt noch einfacher per Tastaturkommandos ferngesteuert werden. Die neue Makrosprache ist allerdings nicht abwärtskompatibel. Existierende Scripts lassen sich aber leicht an die neue Fernsteuerungs-Makrosprache anpassen.
- Implementierung der Funktion EXT: Über EXT lassen sich externe DLL-Funktionen über ein standardisiertes RapidBATCH Funktionsformat aufrufen - somit können in anderen Programmiersprachen - wie beispielsweise C oder Delphi - erstellte Funktionen über DLL-Aufrufe auch direkt aus einem RapidBATCH-Script angesteuert werden, was völlig neue Möglichkeiten eröffnet. Dieses Feature ist bereits undokumentiert seit RapidBATCH 5.0.3 implementiert, wurde aber bis zu dieser Version ausgiebig erprobt und getestet.
- Um auch auf Fensterelemente in RapidBATCH Dialogen über externe DLL-Funktionen zugreifen zu können, können bei GETDIALOG jetzt auch die Attribute Hwnd (für das interne Window-Handle eines Widgets) und TYPE (zum ermitteln des Widget-Typs) zu jedem Widget abgefragt werden.
- Fehlerhaftes GETFILEATT in EXE-Programmen behoben.
- Fehler bei gleichnamigen globalen und lokalen Variablen in EXE-Dateien behoben
- Fehlerbehebung bei der Suche im RapidBATCH Builder: Der Builder hängt sich jetzt nicht mehr auf, wenn ein bereits gesuchter String ersetzt wurde, und nicht mehr gefunden wird.
- Zahlreiche kleinere Fehler aus den erweiterten RapidBATCH Funktionsbibliotheken bereinigt

Mit der Version 5.1 wird RapidBATCH daher noch produktiver und effizienter als je zuvor.

## Wer entwickelt RapidBATCH?

RapidBATCH ist eine wachsende Programmiersprache, die stetig fort- und weiterentwickelt wird.

Entwickler von RapidBATCH ist die kleine Softwareschmiede J.M.K S.F. Software Technologies aus Unna (nahe Dortmund in Nordrhein-Westfalen), deren Inhaber und Haupt-Entwickler der Jungunternehmer Jan Max Meyer ist.

Wir haben es uns zum Ziel gesetzt, neue Programmiersysteme zu entwickeln, die einfachste Handhabung, höchste Qualität und exzellente Performance bieten. Mit der vorliegenden, neuen Version 5 von RapidBATCH, haben wir dieses Ziel mehr als nur erreicht.

Wir hoffen sehr, dass RapidBATCH Ihren Ansprüchen genügt und Sie es bestmöglich an- und verwenden können. Trotzdem ist uns Ihre Meinung wichtig: Wenn Sie noch Anregungen oder Kritiken für kommende RapidBATCH-Versionen haben, können Sie uns gerne eine e-Mail mit Ihren Ideen zukommen lassen.

Weitere Informationen zu J.M.K S.F. Software Technologies sowie unseren Produkten und unserer Kontakt-Adresse finden Sie auf der J.M.K S.F. Homepage unter <http://www.jmksf.de>

## Die Grundlagen

### Ein einfaches Script

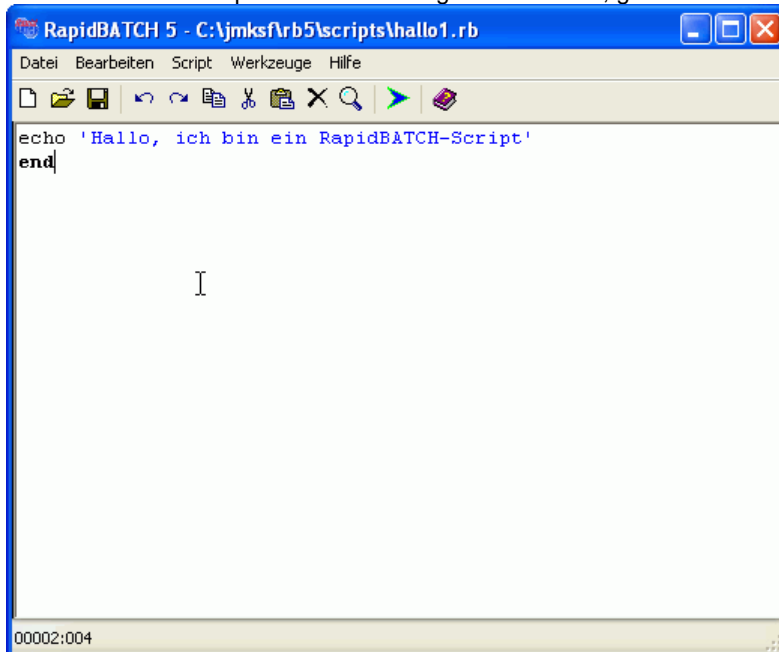
Bereits in der Einleitung haben wir Ihnen ein einfaches, zweizeiliges Script vorgestellt, welches den einfachen Aufbau eines in RapidBATCH programmierten Programms darstellen sollte:

```
echo 'Hallo, ich bin ein RapidBATCH-Script'  
end
```

Um dieses Script auszuführen, müssen wir es erst einmal editieren und als RapidBATCH-Scriptdatei mit der Dateierweiterung .RB speichern.

Dazu sollten Sie den mitgelieferten Scripteditor RapidBATCH Builder benutzen, welcher Ihnen das komfortable Editieren von RapidBATCH-Scripts gestattet. Den Builder können Sie am einfachsten über das Startmenü (Start -> Programme -> RapidBATCH Personal Edition/RapidBATCH Professional Edition -> RapidBATCH 5.0) starten.

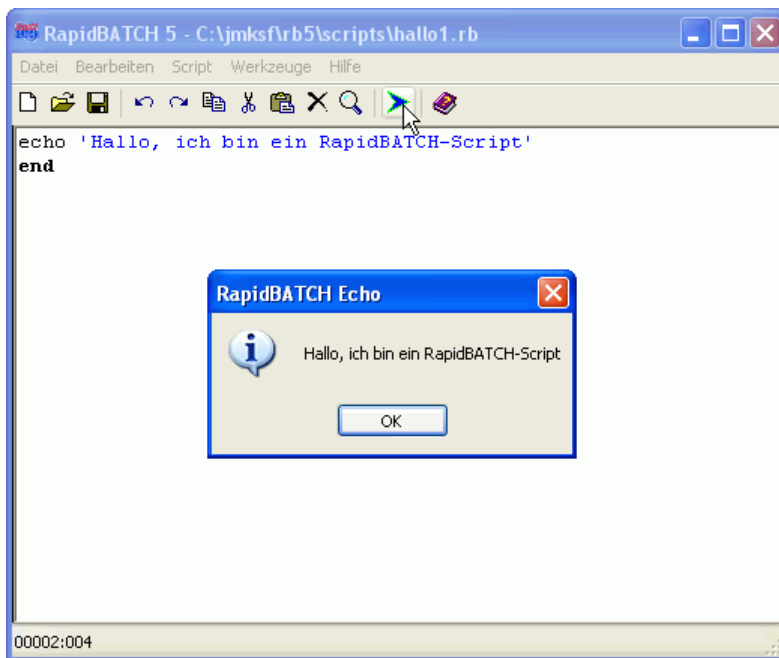
Nachdem Sie den RapidBATCH Builder gestartet haben, geben Sie dort den oben abgebildeten Quelltext ein.



Eingabe des Script-Codes im RapidBATCH Builder

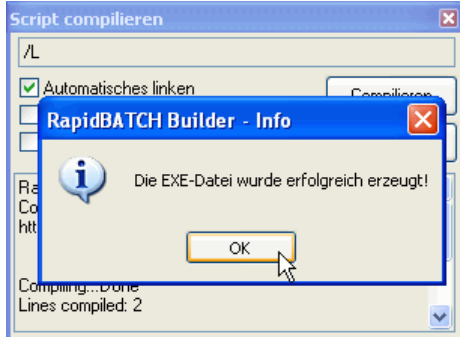
Speichern Sie anschließend das Script in einem Verzeichnis Ihrer Wahl unter dem Dateinamen HELLO.RB (oder auch einem beliebigen anderen Dateinamen) ab. Nun kann das Script gestartet werden, indem Sie F5 drücken oder den Button "Script starten" in der Toolbar anklicken.





Ausführung des Scripts aus dem RapidBATCH Builder heraus  
Herzlichen Glückwunsch zu Ihrem ersten, lauffähigen Script in RapidBATCH!

Wenn Sie im Besitz der RapidBATCH Professional Edition sind, können Sie Ihr Script durch Auswahl des Menüpunktes "Script" und dann "Compilieren" in eine ausführbare EXE-Programmdatei übersetzen. Diese EXE-Programmdatei kann als eigenständiges Programm auf jedem PC mit einem Betriebssystem ab Windows 95 ausgeführt werden. Das Programmicon der EXE-Datei ist mit einem individuellen Icon austauschbar, welches im Compiler-Optionsdialog ausgewählt werden kann.



Der RapidBATCH Script-Compiler zum Überführen einer .RB-Scriptdatei in ein lauffähiges EXE-Programm



Nach der Compilierung findet sich neben der .RB-Datei auch eine ausführbare .EXE-Datei unter selbem Namen. Nachdem Sie nun das Script ausgeführt und compiliert haben, kommen wir zu Erklärung des obigen Beispiels: Die Anweisung ECHO gibt, wie Sie sicherlich schon vermutet haben, das Meldungsfenster aus. Als Parameter erwartet diese Anweisung einen Wert, nämlich den Meldungstext, der in dem Meldungsfenster angezeigt werden soll, also "Hallo, ich bin ein RapidBATCH-Script". Bei einigen Anweisungen muss der Parameter nicht unbedingt zwischen zwei Hochkommata ('...'), wie es hier der Fall ist, eingeschlossen werden, dies ist jedoch ein reiner Sonderfall, der nur bei Anweisungen oder Funktionen mit einem Parameter erlaubt ist. Im Allgemeinen sollten Sie Werte aber möglichst immer in Hochkommata schreiben, da die klassische Schreibweise (Werte ohne Hochkommata anzugeben) nicht mehr dem modernem RapidBATCH-Sprachstandard entspricht.

Die END-Anweisung, welche der ECHO-Anweisung in einer neuen Zeile folgt, markiert das Ende des Scripts. Sie muss zwar nicht zwingend angegeben werden, gehört aber zur sauberen Programmierung in RapidBATCH einfach dazu.

Ob Sie die Befehle gross oder klein schreiben, ist völlig egal, RapidBATCH macht keine Unterschiede zwischen "ECHO", "eChO" und "Echo". Es ist also Ihnen überlassen, wie Sie die Befehle schreiben möchten. Ich bevorzuge die Kleinschreibweise, wie im obigen Beispiel. Leerzeichen können überall im Code und zwischen Parametern eingesetzt wer-



den, ebenso auch Tabulatoren. Das Ende der Zeile, also der Zeilenumbruch, markiert gleichzeitig das Ende der Anweisung, d.h. Sie müssen für jede neue Anweisung Ihres Scripts eine neue Zeile beginnen.

Bevor wir jetzt richtig loslegen mit den mächtigen Funktionen und schier unbegrenzten Möglichkeiten, die Ihnen RapidBATCH bietet, möchte ich vorweg noch auf die Verwendung von Kommentaren hinweisen. Kommentare kann man als Notizen im Quelltext ansehen, um die Verwendung bestimmter Operationen oder Vorgehensweisen zu Dokumentieren oder um zu beschreiben, was eine oder mehrere Anweisungen für ein Ziel verfolgen.

Kommentare helfen Ihnen, anderen Programmierern oder sogar dem Anwender, bestimmte Abläufe im Script nachzuvollziehen und besser bzw. schneller zu verstehen. Ich kommentieren daher den Quellcode der Beispielscripts in diesem Programmierhandbuch immer da, wo es mir wichtig erscheint; Das bedeutet nicht, dass jede einzelne Zeile explizit kommentiert wird oder werden muss, sondern nur die wichtigen bzw. kritischen Programmpunkte.

Um in RapidBATCH einen Kommentar einzuleiten, bedarf es der REM-Anweisung (was soviel heißt wie das englische Wort "remark" für "Bemerkung"). Diese Anweisung leitet einen einzeiligen Kommentar ein, Beispiel wäre hier (bezogen auf unser erstes Beispielscript):

```
rem Ein einfaches Programm mit Ausgabe eines Textes in einem Meldungsfenster
echo 'Hallo, ich bin ein RapidBATCH-Script'

rem Programmende
end
```

Kommentare können nicht direkt hinter Programmieranweisungen folgen, sondern werden intern vom Interpreter bzw. Compiler als eigenständige "Anweisung" angesehen; Daher muss jeder Kommentar immer in eine separate Zeile geschrieben werden.

Im nächsten Kapitel steigen wir nun richtig in die Materie der Programmierung von RapidBATCH-Scripts ein. Wenn Sie mit der Verwendung des RapidBATCH Builders noch nicht so vertraut sind, testen Sie die verschiedenen Funktionalitäten ruhig einmal aus.

## Variablen

Nachdem Sie nun die erste Hürde, das erste eigene Script, überwunden haben, geht's erstmal richtig los!

Wir werden nun ein Script entwickeln, das mit Variablen, also nicht konstanten Werten arbeitet. Diese so genannten Variablen spielen nämlich in der Programmierung von Computern eine sehr wichtige Rolle, da sie den Programmablauf beeinflussen und steuern. Der Ablauf des Programms kann also von diesen variablen Werten gesteuert und verändert werden.

Variablen sind also Speicherbereiche, denen ein Wert zugewiesen werden kann. Das nun folgende Script soll einmal den Einsatz von Variablen demonstrieren. Dazu erstellen Sie im RapidBATCH Builder eine neue, leere Script-Datei und geben folgenden Quellcode ein:

```
rem Eingabe des Benutzernamens
inputbox [benutzer] = 'Eingabe', 'Hallo, wie ist Ihr Name?', ''

rem Ausgabe des Benutzernamens
echo [benutzer]
end
```

Dieses Programm fragt mit Hilfe des INPUTBOX-Dialogs zuerst den Namen des Anwenders ab. Dieser wird in die Variable mit dem Namen [benutzer] gespeichert, weil diese vor dem Gleichheitszeichen (=) der INPUTBOX steht. Die INPUTBOX ist eine so genannte Funktion, genaueres dazu erläutere ich jedoch später.

Variablen werden in RapidBATCH immer von eckigen Klammern umschlossen, und können überall eingesetzt werden, wo auch konstante Werte möglich sind.

In der darauf folgenden ECHO-Anweisung geben wir die Variable wieder aus. Dies geschieht durch einfaches Angeben des Variablennamens anstatt eines konstanten Wertes, wie wir es im ersten Beispiel getan haben. Wir geben hier also anstatt eines festen Wertes den völlig frei vom Anwender des Scripts definierten Wert der Variablen [benutzer] aus, d.h. die Ausgabe, die im Meldungsfenster erscheint, ist abhängig von dem, was der Benutzer im Eingabefenster eingegeben hat.

Variablen in RapidBATCH können sowohl Texte, so genannte Strings (zu deutsch: "Zeichenketten"), als auch Zahlen speichern, berechnen und manipulieren. Die Konvertierung zwischen den verschiedenen Datentypen findet dabei intern statt, so dass sich der Programmierer darum nicht explizit kümmern muss.

Seit RapidBATCH 4.0 müssen Variablen auch nicht mehr explizit mit der DEC-Anweisung (was soviel bedeutet wie "Declare", zu deutsch "Deklariere") vor ihrer Verwendung deklariert werden. DEC spielt aber trotzdem noch eine sehr wichtige Rolle, auf die wir aber später erst eingehen möchten. Wenn Sie einer Variablen, die nicht existiert, einen Wert zuweisen, wird diese also automatisch im Speicher erzeugt. Die Variable ist dann solange im Speicher, bis das Script beendet wird.

Um einer Variable einen Wert zuzuweisen, genügt eine einfache Zuweisung in der Form

```
[variable] = 'Wert'
```

z.B.

```
[benutzer] = 'Anna'
```

oder, um einer Variablen den Wert einer bereits existierenden Variablen zuzuweisen:

```
[benutzer] = [name]
```

Bei Variablenzuweisungen lassen sich Werte auch verketteten und berechnen. Zu diesem Zweck stellt RapidBATCH sechs verschiedene Operatoren zur Verfügung, die zur Verkettung und Berechnung von Werten verwendet werden können. Folgendes Script verbindet beispielsweise mit Hilfe des Join-Operators (#) eine Variable mit einem konstanten String und weist das Ergebnis dieser Zuweisung der Variablen [ausgabe] zu:

```
[ausgabe] = 'Aha, Ihr Name ist also ' # [benutzer]
```

Diese Verkettung zweier Werte kann auch direkt als Parameter einer Anweisung angegeben werden, Beispiel wäre hier

```
echo 'Aha, Ihr Name ist also ' # [benutzer]
```

Mit den Operatoren +, -, \*, / und ~ für Addition, Subtraktion, Multiplikation, Division und Modulo (Restwert) können Variablen und Werte auch berechnet werden. Wichtig ist hierbei, dass die Variablen einen numerischen Wert enthalten. Enthält der Wert einer Variablen z.B. den Text "Hallo" anstatt einer Zahl, so wird er bei einer Berechnung als "0" interpretiert.

Folgendes Script zeigt, wie man mit Hilfe des Additions-Operators (+) eine Addition in RapidBATCH durchführt:

```
[a] = '2'  
[b] = '5'  
[ergebnis] = [a] + [b]  
echo [a] # ' + ' # [b] # ' ergibt ' # [ergebnis]  
end
```

Da man Werte auch direkt berechnen kann, hat folgendes Script genau dieselbe Ausgabe:

```
[a] = '2'  
[b] = '5'  
echo [a] # ' + ' # [b] # ' ergibt ' # ([a] + [b])  
end
```

Hier weisen wir den Wert nicht vorher einer Variablen zu, sondern lassen ihn direkt bei der Ausgabe berechnen, also da, wo das Ergebnis benötigt wird.

RapidBATCH 5 ist hierbei sehr flexibel; Es können somit beispielsweise auch ganze Formeln durch Klammersetzung in einem Rutsch berechnet werden:

```
[a] = '5'  
[b] = '12'  
echo [a] * ([b] + '4' * ([a] - '2')) / [b]  
end
```

Obiges Script würde zur Ausgabe des Wertes "10" führen!

Zudem bietet RapidBATCH bei Zuweisungen eine verkürzte Schreibweise an, um z.B. einen Wert auf sich selber aufzuaddieren.

```
[a] + '5'
```

ist äquivalent zur bisher bekannten Zuweisung

```
[a] = [a] + '5'
```

oder auch eine Quadrierung

```
[a] * [a]
```

Diese Kurzform für Zuweisungen ist mit allen RapidBATCH-Operatoren möglich, und spart viel Tipparbeit.

## Werte manipulieren

Da Sie nun die Verwendung von Variablen kennengelernt haben, gehen wir noch einen Schritt weiter. Variablen sind nämlich nicht nur dazu da, um darin Werte abzulegen, sondern auch, um mit diesen Werten zu arbeiten (sie zu verarbeiten).

Zu diesem Zweck stellt Ihnen RapidBATCH so genannte build-in Funktionen bereit. "Build-in" bedeutet, dass diese Funktionen fest in den RapidBATCH-Standardbefehlssatz eingebaut sind und für die meisten, einfachen Operationen schon ausreichen. Diese build-in Funktionen bieten aber auch die ideale Basis, um später eigene, individuelle Funktionen oder Prozeduren mit individuellen Eigenschaften und Programmverhalten zu entwickeln, wozu wir allerdings erst später kommen werden.

Funktionen haben die Eigenschaft, dass sie immer einen Wert zurückgeben; Anders als bei vielen anderen Programmiersprachen ist es in RapidBATCH gang und gebe, dass die Ergebnisvariable, die den Rückgabewert der Funktion erhält, hinter und nicht vor dem Funktionsnamen angegeben wird.

Ein Beispiel zur Anwendung einer Funktion zur Stringmanipulation ist folgendes Script, das den vom Benutzer eingegebenen Wert mit Hilfe der Funktion UPVAR in Großbuchstaben konvertiert:

```
rem Eingabe des Benutzernamens
inputbox [benutzer] = 'Eingabe', 'Guten Tag, wie ist Ihr Name?', ''

rem Benutzernamen in Grossbuchstaben konvertieren
upvar [benutzer] = [benutzer]

rem Ausgabe des Benutzernamens
echo 'Ihr Name in Großbuchstaben lautet: ' # [benutzer]
end
```

Funktionen haben i.d.R. mindestens einen Parameter, hier ist es der Text, der in Großbuchstaben konvertiert werden soll. Hat eine Funktion mehrere Parameter, so werden diese durch Kommata voneinander getrennt. Wenn Sie später selbst Ihre eigenen Funktionen programmieren, sind auch parameterlose Funktionen möglich!

Alle build-in Funktionen von RapidBATCH zur Stringmanipulation beginnen entweder mit einem "Get" oder hören mit einem "Var" auf; Dies ist kein direkter Standard (wenn Sie eigene Funktionen programmieren ist dies völlig irrelevant!), nur wurden und werden alle neu hinzugekommenen und hinzukommenden build-in Funktionen an diese alte Namensumgebung, die bereits seit RapidBATCH 1.0 bekannt sind, angelehnt.

Insgesamt bietet RapidBATCH 12 solcher build-in Funktionen, diese heißen UPVAR, LOWVAR, TRIMVAR, GETCHR, GETASC, COPYVAR, GETCHARAT, GETTOK, GETPOS, REPLACEVAR, CNTVAR und RANDVAR. Ihre Verwendung und praktische Beispiele können in der Sprachreferenz dieses Handbuches nachgeschlagen werden.

Es gibt in RapidBATCH natürlich eine Vielzahl von anderen build-in Funktionen, wie z.B. den Dialog INPUTBOX - auch eine Funktion - dessen Rückgabewert der eingegebene String ist. Die oben aufgeführten build-in Funktionen haben allerdings den Zweck, dass Sie einen bzw. mehrere Eingabewerte verarbeiten und einen daraus resultierenden Wert zurückgeben.

## Bedingungen und Schleifen

Nachdem Sie nun die äusserst komplexen Möglichkeiten der variablen Programmierung kennen gelernt haben, kommen wir zu einem weiteren, wichtigen Teil der Ereignis-orientierten Programmierung, den Bedingungen.

Bisher haben wir nämlich nur Scripts geschrieben, die kontinuierlich von vorne nach hinten durchlaufen. Solche Programme sind mehr oder minder uninteressant für die Entwicklung von Software-Programmen. Doch in RapidBATCH gibt es verschiedene Möglichkeiten, variable Werte zu prüfen und den Programmablauf somit bedingt zu gestalten.

Beginnen wir mit einem einfachen Script, das ein Passwort solange abfragt, bis der Benutzer das korrekte Wort eingibt. Zu diesem Zweck verwenden wir die Schleifenanweisungen REPEAT und UNTIL zur Konstruktion einer Schleife. Eine Schleife ist ein Block von Anweisungen, der solange wiederholt wird, bis eine gegebene Bedingung erfolgreich geprüft wird. Ist dies der Fall, wird die Schleife verlassen und das Script fortgeführt.

Der Code für eine Passwort-Abfrage:

```
rem Schleife solange wiederholen, bis Passwort "hallo" eingegeben wurde
repeat
    pwdbox [password] = 'Login', 'Bitte geben Sie Ihr Kennwort ein:', ''
until [password] = 'hallo'
echo 'Das Kennwort war richtig.'
end
```

Wie Sie sehen werden, wird das PWDBOX-Eingabefenster (eine INPUTBOX mit verstecktem Eingabefeld für Passwörter) solange wieder angezeigt, bis der Benutzer das richtige Passwort, in diesem Fall "hallo", eingegeben hat. Bei dieser Bedingung handelt es sich um eine Gleichheitsbedingung, d.h. sie liefert das Ergebnis WAHR, wenn beide zu vergleichende Werte exakt den selben Wert beinhalten. Neben dem Gleichheitsoperator (=) kann man bei Bedingungen in RapidBATCH auch die Bedingungsoperatoren ungleich (!), größer-als (>), kleiner-als (<) sowie größer-gleich (>=) und kleiner-gleich (<=) verwenden, um eine Bedingung zu konstruieren. Die letzten vier können in RapidBATCH 5 auch mit Strings benutzt werden, da z.B. der Buchstabe 'A' kleiner als der Buchstabe 'B' ist. In RapidBATCH 4.x waren größer-als und kleiner-als Bedingungen nur mit numerischen Werten möglich.

Zudem empfehle ich, den Code, welchen Sie zwischen die Anweisungen REPEAT und UNTIL schreiben, mit Leerzeichen oder Tabs einzurücken, denn dies verbessert die Lesbarkeit des Codes erheblich und trägt zur schnelleren Fehlersuche bei, da der Code, der zur Schleife gehört, durch die Einrückung direkt hervorsticht.

Schleifen sind jedoch nur eine Möglichkeit der bedingten Programmierung in RapidBATCH. Die andere, viel häufiger verwendete Methode ist die IF-Anweisung, die zusammen mit den Anweisungen ELSEIF, ELSE und ENDIF in den verschiedensten Variationen eingesetzt werden kann. Folgende Beispiele demonstrieren einmal ein paar Einsatzfälle für IF-Konstruktionen.

Eine einfache IF-Anweisung (zu deutsch: "Wenn") wäre z.B.

```
if [eingabe] = 'hallo' echo 'Das eingegebene Wort wurde erfolgreich auf "hallo" geprüft'
```

Hier folgt direkt hinter der IF-Anweisung und der dazugehörigen Bedingung die auszuführende Anweisung. Möchte man zwei oder mehr Anweisungen hinter einem IF ausführen, gibt man diese in einem so genannten IF-Block an:

```
if [eingabe] = 'hallo'
    echo 'Das eingegebene Wort wurde erfolgreich auf "hallo" geprüft'
    [x] + '2'
    if [x] > '32' halt
endif
```

Der Zeilenumbruch hinter der Bedingung weist den Interpreter bzw. Compiler an, dass er, wenn die Bedingung erfolgreich ist, den gesamten Code bis zum nächsten ENDIF (oder ELSEIF/ELSE) ausführen soll. Die HALT Anweisung, die hier verwendet wird, beendet das Script.

Wenn man mehrere verschiedene Bedingungen in unterschiedlichen IF-Blöcken prüfen möchte, verwendet man die ELSEIF-Anweisung:

```
if [eingabe] = 'hallo'
    echo 'Das eingegebene Wort wurde erfolgreich auf "hallo" geprüft'
    [x] + '2'
    if [x] > '32' halt
elseif [eingabe] = 'tschüss'
    echo 'Auf wiedersehen!'
```

```
halt  
endif
```

Wenn keine der Bedingungen eintritt, also alle IF/ELSEIF-Bedingungen unwahr sind, so kann man auch einen bestimmten Anweisungsblock in einem ELSE (zu deutsch: "Ansonsten") zusammenfassen, der eben nur dann ausgeführt wird, wenn alles andere fehlschlägt:

```
if [eingabe] = 'hallo'  
    echo 'Das eingegebene Wort wurde erfolgreich auf "hallo" geprüft'  
    [x] + '2'  
    if [x] > '32' halt  
elseif [eingabe] = 'tchüss'  
    echo 'Auf wiedersehen!'  
    halt  
else  
    echo 'Eingabe ungenau.'  
    goto neustart  
endif
```

Manchmal ist es auch von Nöten, mehr als einen Vergleich bei einer REPEAT...UNTIL oder IF/ELSEIF-Bedingung zu prüfen. Zu diesem Zweck stellt RapidBATCH zwei so genannte logische Verknüpfungsoperatoren bereit, die UND-Verknüpfung ("&"-Operator) sowie die ODER-Verknüpfung ("|" -Operator). Diese Operatoren werden verwendet, um zwei (oder mehr) Vergleiche logisch zu einer gesamten Bedingung zu verknüpfen.

Bei der ODER-Verknüpfung muss eine der verknüpften Bedingungen wahr sein, damit die Abfrage erfolgreich geprüft wird.

Beispiel ist folgendes Script. Es gibt nur eine Willkommensnachricht aus, wenn der eingegebene Name "Stefan" oder "Peter" ist.

```
inputbox [name] = 'Log in', 'Wie heißt du?', ''  
if [name] = 'Stefan' | [name] = 'Peter'  
    echo 'Herzlich Willkommen!'  
else  
    echo 'Du bist nicht Stefan oder Peter!'  
endif  
end
```

Anders ist es bei der UND-Verknüpfung: Hier müssen beide Bedingungen wahr ergeben, damit die Abfrage als erfolgreich geprüft wird.

```
inputbox [name] = 'Nachname', 'Wie ist dein Nachname?', ''  
inputbox [vorname] = 'Vorname', 'Wie ist dein Vorname?', ''  
  
if [name] ! '' & [vorname] ! ''  
    echo 'Hallo ' # [vorname] # ' ' # [name]  
else  
    echo 'Sie haben nicht alle Werte eingegeben!'  
endif  
end
```

Natürlich können auch UND- und ODER-Verknüpfungen beliebig hintereinander geschaltet werden, nur müssen Sie darauf achten, dass die gegebenen Bedingungen auch wirklich erreicht werden können.

## Sprünge und Subs

Ein bewährtes Mittel, den Programmablauf strukturiert zu gestalten, ist der Sprung zu Codelabels und Unterprogrammen, den so genannten Subs. Da allerdings die meisten Probleme mit eigenen Prozeduren und Funktionen und den Schleifen-Anweisungen REPEAT und UNTIL gelöst werden können, wird dieses Thema nur einmal kurz angesprochen.

Um mit Codelabels bzw. Unterprogrammen zu arbeiten stehen Ihnen die Anweisungen GOTO, GOSUB und RESUME zur Verfügung. Mit GOTO können Sie vorwärts oder rückwärts durch das Script "springen", d.h. Sie können mit der Programmausführung einfach an einer anderen Position im Script fortfahren. Diese GOTO-Sprünge werden von vielen Programmierern oft verhasst, weil sie den Code unschön und unübersichtlich machen, doch manchmal sind solche Sprünge einfach unerlässlich und bringen auch viele Vorteile mit sich (z.B. höhere Performance).

Einfachstes Fallbeispiel ist z.B. folgendes Script. Wir haben drei Meldungsfenster, von denen jedoch nur das erste und das dritte angezeigt werden. Nach dem ersten ECHO springen wir hier einfach via GOTO zu dem Label %ende. Labels, auch Sprungmarken genannt, sind direkt im Code durch ein vorangestelltes Prozent-Zeichen (%) gekennzeichnet und können jeden beliebigen Namen haben, auch mit Leer- und Sonderzeichen.

```
echo 'Meldung 1'
goto 'ende'
echo 'Meldung 2'
%ende
echo 'Meldung 3'
end
```

Ausserdem gibt es seit RapidBATCH 4.0 auch die Möglichkeit, sehr einfache Unterprogramme, so genannte Subs, zu entwickeln. Dies sind keine wirklichen Prozeduren wie solche, auf die ich später noch genauer eingehen werde, sondern Code-Blöcke, die mit einem Label beginnen und mit einer RESUME-Anweisung enden. Bei Ausführung von RESUME wird das Script hinter dem letzten GOSUB-Aufruf fortgeführt, d.h. das Unterprogramm kann von verschiedenen Positionen aus aufgerufen werden und wird nach dem RESUME wieder an der jeweiligen Stelle fortgeführt, von der es aufgerufen wurde. Folgendes Beispiel nimmt die Zahlen 1 bis 10 zum Quadrat.

```
[i] = '1'
repeat
    gosub 'quadrat'
    [i] + '1'
until [i] > '10'
halt

rem Sub: quadrat
%quadrat
[x] = [i] * [i]
echo [i] # ' * ' # [i] # ' = ' # [x]
resume
end
```

Die GOSUB-Anweisung springt hier, genau wie GOTO, zu einem Label (hier: quadrat).

Beachten Sie bitte, dass man diese Prozeduraufrufe auch beliebig ineinander verschachteln kann, d.h. Subs können wiederum andere Subs oder sich selbst rekursiv aufrufen. Bestes Beispiel wäre hier das folgende Script:

```
echo 'Meldung 1'
gosub 'test1'
echo 'Meldung 2'
halt

%test1
echo 'Meldung 3'
gosub 'test2'
resume

%test2
echo 'Meldung 4'
gosub 'test3'
resume

%test3
echo 'Meldung 5'
gosub 'test4'
resume

%test4
echo 'Meldung 6'
resume

end
```

Die Meldungsfenster erscheinen in folgender Reihenfolge: 1, 3, 4, 5, 6, 2!

Seit RapidBATCH 5.0 sind auch dynamische Sprünge via GOTO und GOSUB möglich. Das bedeutet, dass bei GOTO bzw. GOSUB als Label auch eine Variable oder ein zusammengesetzter Wert angegeben werden kann, zu dem gesprungen wird.

Beispiel:

```
confirm [answer] = 'Ja oder Nein?'
goto [answer]

%0
echo 'Sie haben JA angeklickt!'
halt

%-1
echo 'Sie haben NEIN angeklickt!'
end
```

Interessant werden diese dynamischen Sprünge u.a. noch etwas später, wenn es um benutzerdefinierte Dialoge geht. Sie sind jedoch mit Vorsicht zu genießen, da ein falsch zusammengesetzter Labelname zum sofortigen Verlassen des Scripts führt, wenn das Label nicht im Script definiert ist.

Labels sollten aber nur dann verwendet werden, wenn dies zwingend notwendig ist. Vermehrt soll in RapidBATCH 5 mit Prozeduren und Funktionen gearbeitet werden, welche Sie, wie bereits erörtert, später genauer kennenlernen werden.

## Arrays

Kommen wir nun zu einer revolutionären Neuerung in RapidBATCH 5, den Arrays!

Bei einem Array (engl. für Anordnung, Aufstellung, Datenfeld) handelt es sich um eine Gruppe von Variablen, die unter einem gemeinsamen Variablen-Namen gespeichert werden. Arrays werden benötigt, um einerseits eine große Anzahl von Variablen mit geringem Aufwand zu definieren und um andererseits die Voraussetzung zu schaffen, direkt auf ein bestimmtes Variablenelement durch Angabe eines Index-Wertes zuzugreifen.

Arrays bieten also die Möglichkeit, unter dem Namen einer einzigen Variablen schier unendlich viele, verschiedene Werte in sogar mehrfach verschachtelten Dimensionen zu speichern. Ein Array kann man sich wie eine Tabelle im Speicher vorstellen, in der jede Zeile einen Wert enthält, welcher über den soeben beschriebenen Index-Zähler adressiert werden kann.

Viele Programmiersprachen bieten sowohl statische Arrays (Arrays, die in ihrer Größe nicht variabel sind) als auch dynamische Arrays (Arrays, die zur Laufzeit definiert werden und dynamische Größen und Dimensionen annehmen können). RapidBATCH ermöglicht nur den Einsatz von dynamischen Arrays, was eine unglaublich hohe Flexibilität zulässt; Der Programmierer braucht sich nämlich in keiner Weise mit der internen Speicherverwaltung des Arrays auseinanderzusetzen - RapidBATCH erledigt die Dimensionierung und Erweiterung von Arrays intern völlig automatisch.

Ein einfaches Beispiel für die Verwendung eines Arrays demonstriert folgendes Script. Es speichert 10 unterschiedliche Namen in einem Array, welches unter als [namen] deklariert wird. In der REPEAT...UNTIL-Schleife weiter unten werden die Namen von 1 bis 10 durchlaufen und angezeigt, wobei wir eine Zählervariable, [i], zum Hochzählen des Indexes verwenden.

```
dec [namen:'10']
[namen:'1'] = 'Jan'
[namen:'2'] = 'Anna'
[namen:'3'] = 'Stefan'
[namen:'4'] = 'Matthias'
[namen:'5'] = 'Sebastian'
[namen:'6'] = 'Daniel'
[namen:'7'] = 'Uli'
[namen:'8'] = 'Andreas'
[namen:'9'] = 'Volker'
[namen:'10'] = 'Marco'

[index] = '0'
repeat
  [index] + '1'
  echo 'Name an Position ' # [index] # ' ist: ' # [namen:[index]]
until [index] = '10'
end
```

Die folgende Tabelle veranschaulicht die Speicheransicht des Arrays [namen] im obigen Beispiel:



Index:	Wert:
1	Jan
2	Anna
3	Stefan
4	Matthias
5	Sebastian
6	Daniel
7	Uli
8	Andreas
9	Volker
10	Marco

Arrays können in RapidBATCH auf verschiedene Arten definiert, dimensioniert und re-dimensioniert werden. Ein einfaches Array mit vorerst 10 Elementen, wie das, welches wir in unserem letzten Beispiel verwendet haben, kann man z.B. mit der DEC-Anweisung fest vordeklarieren:

```
dec [meinArray:'10']
```

In RapidBATCH ist es völlig frei, wie der Index (in diesem Fall der feste Wert '10') eines Arrays angegeben wird, er kann neben der oben gezeigten Variante auch in den folgenden Formen angegeben werden:

```
dec [meinArray'10']  
dec [meinArray '10']  
dec ['10'meinArray]  
dec [mein'10'Array]
```

Auch wenn letztere Schreibweise sehr merkwürdig aussieht, sie ist eine gültige Index-Angabe, und das Array wird unter dem Namen "meinArray" deklariert!

Der Index kann selbstverständlich auch völlig dynamisch über eine Variable angegeben werden:

```
[index] = '10'  
dec [meinArray:[index]]  
dec [meinArray[index]]  
dec [mein[index]Array]
```

Arrays können auch durch direkte Zuweisung eines Wertes zu irgend einem Index deklariert bzw. erzeugt werden; Das bedeutet auch, dass Sie jede beliebige, in Ihrem RapidBATCH-Script verwendete Variable durch eine einfache Zuweisung eines Wertes zu einem Index in ein Array umwandeln können. Folgendes Beispiel erzeugt ein Array mit drei Werten durch die direkte Zuweisung der Werte, ohne vorherige, explizite Deklaration. Die Elemente des Arrays werden dabei rückwärts ausgegeben, d.h. der letzte Eintrag zuerst, wie aus der Schleife ersichtlich ist.

```
[meinArray:'1'] = 'Ich bin der erste Eintrag in dem Array'  
[meinArray:'2'] = 'Ich bin der zweite Eintrag in dem Array'  
[meinArray:'3'] = 'Ich bin der letzte Eintrag in dem Array'
```

```
[i] = '3'  
repeat  
    echo [meinArray:[i]]  
    [i] - '1'  
until [i] = '0'  
end
```

Doppelpunkte und Kommata werden dabei im Variablennamen einfach ignoriert, d.h. sie sind nicht zwingend notwendig. Um die Anzahl der Elemente, die je Dimension in einem Array existiert, zu ermitteln, verwendet man die Dimensionangabe 0:

```
[meinArray:'1'] = 'Ich bin der erste Eintrag in dem Array'  
[meinArray:'2'] = 'Ich bin der zweite Eintrag in dem Array'  
[meinArray:'3'] = 'Ich bin der letzte Eintrag in dem Array'  
echo 'Das Array beinhaltet momentan ' # [meinArray:'0'] # ' Elemente'  
  
[i] = '0'  
repeat  
    [i] + '1'  
    echo 'Eintrag Nr. ' # [i] # ': ' # [meinArray:[i]]  
until [i] = [meinArray:'0']  
end
```

Wie bereits oben kurz angesprochen, können Arrays auch in verschiedenen, verschachtelten Dimensionen angelegt werden. Eine Dimension ist sozusagen ein untergeordnetes Array eines Array-Elementes. D.h. wir können für jedes Array-Element ein untergeordnetes Array anlegen. Jedes Element dieses untergeordneten Arrays kann wiederum ein untergeordnetes Array beinhalten - das ganze ist in RapidBATCH bis zu 9 Dimensionen möglich, eine immens große Kapazität.

Um mehrdimensionale Arrays anzulegen geht man genauso vor wie wenn man ein einfaches Array deklariert bzw. auffüllt: Man weist einfach einem nicht existierenden Element in irgendeiner Dimension einen Wert zu - RapidBATCH redimensioniert bzw. reserviert sich den nötigen Speicher, der zum Erreichen der angegebenen Dimension nötig ist, automatisch.

Ein Beispiel (wobei die DEC-Anweisung auch hier wieder optional ist!):

```
rem Deklarieren eines Arrays mit 10x10x10 Dimensionen
dec [meinArray:'10','10','10']
[meinArray:'1','1','1'] = 'Hello World'
[meinArray:'1','1','2'] = 'Wert an Index 2'
[meinArray:'1','1','3'] = 'Wert an Index 3'
[meinArray:'1','2','1'] = 'Anderer Wert an Index 1'
[meinArray:'1','2','2'] = 'Anderer Wert an Index 2'
[meinArray:'1','2','3'] = 'Anderer Wert an Index 3'
```

RapidBATCH bietet an dieser Stelle auch noch eine Besonderheit, die bisher keine andere Programmiersprache weltweit aufweist: Werte können in JEDE Dimension eines Arrays gespeichert werden!

Ein konkretes Beispiel:

```
rem Auffüllen des Arrays in verschiedenen Dimensionen
```

```
[meinArray] = 'Basis'
[meinArray:'1'] = 'Erste Dimension, Wert 1'
[meinArray:'1','1'] = 'Zweite Dimension, Wert 1'
[meinArray:'1','1','1'] = 'Dritte Dimension, Wert 1'
[meinArray:'1','1','2'] = 'Dritte Dimension, Wert 2'
[meinArray:'1','2'] = 'Zweite Dimension, Wert 2'
[meinArray:'2'] = 'Erste Dimension, Wert 2'
```

```
rem Ausgabe des Arrays
```

```
echo [meinArray]
echo [meinArray:'1']
echo [meinArray:'1','1']
echo [meinArray:'1','1','1']
echo [meinArray:'1','1','2']
echo [meinArray:'1','2']
echo [meinArray:'2']
```

Speziell für die Wiederverwertung bzw. Auflösung nicht mehr benötigter Arrays und deren Speicherressourcen gibt es in RapidBATCH 5 auch eine Möglichkeit, ein Array komplett zu löschen (z.B. damit es neu aufgefüllt werden kann): Die RESET-Anweisung! RESET tut genau das Gegenteil von der Anweisung DEC zum Deklarieren von Variablen und Arrays. Mit der RESET-Anweisung kann ein ganzes Array oder auch nur eine einzelne Dimension eines Arrays wieder freigegeben, also aus dem Speicher entfernt werden. Anzumerken sei hierbei, dass RESET nicht die Variable des Arrays selbst, sondern nur das Array bzw. den Inhalt der Variablen vom Speicher entfernt und zur erneuten Verwendung freigibt.

Einfaches Beispiel:

```
[meinArray:'1'] = 'Ich bin der erste Eintrag in dem Array'
[meinArray:'2'] = 'Ich bin der zweite Eintrag in dem Array'
[meinArray:'3'] = 'Ich bin der letzte Eintrag in dem Array'
echo 'Das Array beinhaltet momentan ' # [meinArray:'0'] # ' Elemente'
```

```
rem Array löschen
```

```
reset [meinArray]
echo 'Das Array beinhaltet momentan ' # [meinArray:'0'] # ' Elemente'
```

```
[meinArray:'1'] = 'Neuer Inhalt'
[meinArray:'2'] = 'Noch ein neuer Inhalt'
echo 'Das Array beinhaltet momentan ' # [meinArray:'0'] # ' Elemente'
end
```

Die Vorteile, die Arrays in RapidBATCH mit sich bringen, sind sehr vielfältig, vor allem ihre 100%ige Dynamik, die andere Programmier- und Scriptsprachen gar nicht oder nicht in diesem Umfang bieten.

## Prozedurale Programmierung

### Benutzerdefinierte Prozeduren und Funktionen

Eines der effektivsten, neuen Features von RapidBATCH 5 ist die Einführung von benutzerdefinierbaren Prozeduren und Funktionen, ein Feature, das für die moderne, strukturierte Programmierung einfach unerlässlich ist. Mit Prozeduren können Sie die Programmierung Ihrer Scripts erheblich beschleunigen, indem Sie neue Befehle und Funktionen, die genau für Ihre Anwendungen zugeschnitten sind, einfach selbst in RapidBATCH als kleine Unterprogramme implementieren. Eine Prozedur lässt sich also quasi als eine Zusammenfassung von mehreren RapidBATCH-Anweisungen und -Kontrollstrukturen zu einer komplett neuen Anweisung (oder Funktion), die dann in RapidBATCH verwendet werden kann, beschreiben.

Prozeduren werden einmal in einem Script definiert und können anschließend wie ganz normale Anweisungen oder Funktionen gehandhabt werden. Es ist daher sinnvoll, Prozedurdefinitionen immer zu Anfang eines RapidBATCH-Scripts anzugeben. Werden die gleichen Prozeduren in mehreren, unterschiedlichen Scripts verwendet, so können diese in einer separaten Script-Datei als Library-Datei definiert und später zur Verwendung im eigentlichen Script mittels der INCLUDE-Anweisung eingebunden werden (mehr zu diesem Verfahren jedoch später!).

Eine einfache Prozedur (ohne Rückgabewert) wird mit dem Schlüsselwort PROC eingeleitet. Direkt nach PROC folgt der Name, über den die neue Prozedur aufgerufen werden soll. Dieser darf keine Leer- oder Sonderzeichen enthalten. Nach dem Prozedurnamen kann dann, gefolgt von einem Doppelpunkt, eine Liste an Parametervariablen (die pro Variable durch Kommata getrennt sein müssen) angegeben werden, wobei auch eine parameterlose Prozedur definiert werden kann (in solch einem Fall ist kein Doppelpunkt erforderlich!). Nach der PROC-Anweisung folgt dann der entsprechende Code, den die Prozedur enthält. Ausgeleitet wird dieser Code mit der Anweisung ENDPROC. Hinter ENDPROC kann dann eine neue Prozedurdefinition oder normaler Script-Code folgen. Auch hier gilt: Zwischen den Anweisungen PROC und ENDPROC sollte der Script-Code zur besseren Lesbarkeit eingerückt werden!

Ein einfaches Beispiel stellt nun folgendes Script dar. Es definiert eine parameterlose Prozedur "test", welche ein Meldungsfenster ausgeben soll.

```
rem Prozedur-Demoscript
proc test
    echo 'Dies ist die Prozedur TEST!'
endproc

echo 'Vor der Prozedur...'
test
echo '...nach der Prozedur'
echo 'Und noch einmal!'
test
end
```

Wenn wir nun eine Prozedur mit Parametervariablen definieren wollen, müssen diese, wie bereits oben beschrieben, als Parameterliste beim Schlüsselwort PROC angegeben werden. Die dort definierten Variablennamen werden beim Aufruf der Prozedur deklariert und bei der Beendigung der Prozedur wieder aus dem Speicher entfernt. Dies nennt man "lokale Variablendeklaration", da diese Variablen (und auch alle Variablen, die in der Prozedur erzeugt werden) nur zur Laufzeit der Prozedur verwendet werden können. Wir werden später noch genauer auf den Unterschied zwischen lokalen und globalen Variablen eingehen.

Ein einfaches Beispiel für eine Prozedur mit Parametern stellt folgendes Script dar. Es definiert die Prozedur "summe" zur Summierung zweier Werte, die als Parameter übergeben werden. Das Ergebnis der Addition wird als Text in einem Meldungsfenster ausgegeben.

```
rem Prozedur-Demoscript mit Parametern
proc summe: [wert1], [wert2]
    dec [ergebnis]
    [ergebnis] = [wert1] + [wert2]

    echo 'Die Summe von ' # [wert1] # ' und ' # [wert2] # ' ist ' # [ergebnis]
endproc

rem Konstante Parameterwerte
summe '7', '13'
```

```
rem Variable Parameterwerte
inputbox [a] = 'Wert A:', 'Bitte geben Sie eine Zahl ein:', ''
if [a] = '' halt
inputbox [b] = 'Wert B:', 'Bitte geben Sie eine Zahl ein:', ''
if [b] = '' halt

summe [a], [b]
end
```

Aus dieser Prozedur lässt sich jetzt auch eine Funktion definieren, die als Rückgabewert die Summe beider Parameter zurückgibt. Funktionen sind dabei nicht weniger schwer programmierbar wie einfache Prozeduren, nur dass wir hier anstatt der Schlüsselwörter PROC und ENDPROC die Schlüsselwörter FUNC und ENDFUNC verwenden. Um den Wert an die Ergebnisvariable, die beim Prozeduraufruf mit angegeben werden muss, zurückzugeben, verwenden wir hier ausserdem die Anweisung RET, was soviel bedeutet wie "return" (zu deutsch: "Zurückkehren"). RET kann auch, ohne Rückgabewert als Parameter, in einfachen Prozeduren eingesetzt werden, um diese vorzeitig vor dem ENDPROC zu verlassen.

Hier nun unser obiges Script mit einer Funktion "summe".

```
rem Funktions-Demoscript
func summe: [wert1], [wert2]
    dec [ergebnis]
    [ergebnis] = [wert1] + [wert2]

    ret [ergebnis]
endfunc

rem Konstante Parameterwerte
summe [erg] = '7', '13'
echo 'Summe: ' # [erg]

rem Variable Parameterwerte
inputbox [a] = 'Wert A:', 'Bitte geben Sie eine Zahl ein:', ''
if [a] = '' halt
inputbox [b] = 'Wert B:', 'Bitte geben Sie eine Zahl ein:', ''
if [b] = '' halt

summe [erg] = [a], [b]
echo 'Summe: ' # [erg]
end
```

Und schon haben Sie Ihre erste, eigene Funktion in RapidBATCH programmiert! Natürlich ist dies nur ein einfaches Beispiel, denn normalerweise addiert mein zwei Werte direkt im String, also

```
echo 'Summe: ' # '32' + '5'
end
```

Als einfaches Modell für die Programmierung von Prozeduren und Funktionen war dieses Beispiel aber doch recht leicht verständlich. Interessant wird es erst, wenn wir rekursive Prozeduren schreiben oder mit Arrays arbeiten, was in den nächsten Kapiteln folgt.

Eine Sache, die an dieser Stelle aber noch sehr wichtig ist, sind die bereits angesprochenen lokalen Variablen. Man unterscheidet bei der prozeduralen Programmierung immer zwischen globalen und lokalen Variablen. In RapidBATCH sind alle Variablen global, die ausserhalb von Prozeduren, irgendwo im Script-Code, deklariert werden. Auf globale Variablen können alle Prozeduren zur Laufzeit zugreifen, d.h. wenn wir eine globale Variable [test] haben, können wir auch in einer Prozedur auf sie zugreifen. Anders ist es bei Variablen, die als Parameter in Prozeduren und Funktionen definiert oder in den Prozeduren/Funktionen deklariert wurden, diese sind nämlich alle lokal. Man kann auch beim Aufruf einer weiteren Prozedur aus einer Prozedur nicht auf die lokalen Werte der aufrufenden Prozedur zugreifen. Wäre dies möglich, würde ein immenses Chaos ausbrechen, was die Programmierung rekursiver Prozeduren (siehe nächstes Kapitel) sowie die strukturierte, prozedurale Programmierung im Allgemeinen völlig unmöglich und zwecklos machen würde. Der Unterschied zwischen lokalen und globalen Variablen kann manchmal auch zu sehr viel Verwirrung führen, daher rate ich, um Probleme mit globalen Variablen zu vermeiden, immer jede lokale Variable, die nicht als Parameter übergeben wird, in der Prozedur mit Hilfe der DEC-Anweisung explizit zu deklarieren. Einfachstes Fallbeispiel ist folgendes Script:

```
[i] = '10'

proc test
    [i] = '1'
    echo 'i ist hier ' # [i]
endproc

echo 'i ist hier ' # [i]
test
echo 'i ist hier ' # [i]
end
```

Was hier passiert ist recht einfach: Wir wollen [i] in der Prozedur "test" lokal verwenden. Wir weisen [i] einfach einen Wert zu und nehmen an, dass [i] automatisch neu deklariert wird. Dies wird sie jedoch nicht, da [i] bereits schon global existiert, und RapidBATCH annimmt, wir wollten auf die globale Variable [i], die ja bereits existiert, zugreifen. Deshalb müssen wir die Variable [i] nun explizit noch einmal für diese Prozedur deklarieren, indem wir die DEC-Anweisung verwenden:

```
[i] = '10'

proc test
    dec [i]
    [i] = '1'
    echo 'i ist hier ' # [i]
endproc

echo 'i ist hier ' # [i]
test
echo 'i ist hier ' # [i]
end
```

In diesem Falle wird [i] neu, als lokale Variable, in der Prozedur "test" deklariert. Nach Beendigung der Prozedur wird die lokale [i]-Variable aus dem Speicher entfernt, und es kann die "alte", globale [i]-Variable weiter verwendet werden.

Möglicherweise haben Sie noch ein wenig Probleme, den Sinn hinter den lokalen und globalen Variablen richtig zu durchschauen, aber je mehr Sie in RapidBATCH programmieren, desto klarer werden solche Strukturen und Zusammenhänge werden.

Wie Sie professionell die prozedurale Programmierung in RapidBATCH ausnutzen können, wird in den folgenden Kapiteln genauer erläutert. Professionelle Anwendungsbeispiele finden Sie auch in den zahlreichen Beispielprogrammen und Bibliotheksfunktionen, die das RapidBATCH-Programmpaket beinhaltet.

## Rekursive Programmierung von Prozeduren

Bei der rekursiven Programmierung ruft sich eine Prozedur oder Funktion in einem Script selbst wieder auf. Wichtig dabei ist, dass eine Abbruchbedingung gegeben ist, weil sich das rekursive Programm sonst (theoretisch) unendlich oft selbst aufrufen würde. Die rekursive Programmierung bietet daher eine elegante Art und Weise, wie man in sich verschachtelte Operationen schnell und einfach durchführen kann. Diese darf in RapidBATCH natürlich nicht fehlen!

Ein einfaches Beispiel für ein rekursiv ablaufendes Programm ist die Berechnung der Fakultät. Die Berechnung der Fakultät ist mathematisch wie folgt definiert:

- Die Fakultät der Zahl 0 ist definitionsgemäß 1.
- Die Fakultät einer ganzen Zahl, die größer als 0 ist, ist das Produkt dieser Zahl mit der Fakultät der nächstkleineren ganzen Zahl.

Wir werden nun solch ein Script programmieren, welches die Fakultät rekursiv berechnet:

```
rem Funktion fac zur Berechnung der Fakultät
func fac: [x]
  dec [ergebnis]

  if [x] = '0'
    [ergebnis] = '1'
  else
    rem Rekursiver Aufruf von fac
    fac [ergebnis] = [x] - '1'
    [ergebnis] * [x]
  endif

  ret [ergebnis]
endfunc
```

```
rem Erstaufruf der Funktion fac
fac [fakultät] = '4'
```

```
rem Ausgabe des Ergebnisses
echo 'Die Fakultät von 4 ist ' # [fakultät]
end
```

Das ist schon alles! Wie man unschwer erkennen kann, ruft sich die Funktion FAC rekursiv aus sich selber heraus auf - allerdings mit dem dekrementierten (um 1 verminderten) Wert der Variablen [x]. Abbruchbedingung in diesem Fall ist die Prüfung, ob [x] den Wert '0' enthält. Ist dies der Fall, ruft sich die Funktion nicht mehr weiterhin selbst auf, und löst die Rekursion auf.

Da ja bei der Rekursion gleich mehrere Hundert (oder gar Tausend) Aufrufe der selben Funktion möglich sind, befinden sich natürlich alle Aufrufe in einem so genannten Stack ("Stapelspeicher") im Hauptspeicher des Computers. Durch die Abbruchbedingung werden nun alle aufgerufenen Funktionen wieder sorgfältig abgearbeitet und beendet.

Würde man bei unserer Fakultäts-Berechnungsfunktion keine solche Abbruchbedingung definieren, würde es irgendwann zu einem Speicherüberlauf und damit zum Programm- oder gar Systemabsturz kommen, da sich die Funktion so lange aufruft, bis kein Speicher mehr verfügbar ist.

Um den Verlauf der Rekursion unseres Fakultäts-Scripts etwas Verständlicher zu machen, können wir auch noch ein paar ECHO-Meldungen zur Programmverfolgung einfügen:

```
[rekebene] = '0'
```

```
rem Funktion fac zur Berechnung der Fakultät
func fac: [x]
  dec [ergebnis]

  rem Ausgabe der Rekursionstiefe
  echo 'Start in Rekursionsebene ' # [rekebene] # ', Parameterwert ' # [x]
  [rekebene] + '1'

  if [x] = '0'
    [ergebnis] = '1'
  else
    rem Rekursiver Aufruf von fac
    fac [ergebnis] = [x] - '1'
    [ergebnis] * [x]
  endif

  rem Ausgabe der Rekursionstiefe
  [rekebene] - '1'
  echo 'Ende in Rekursionsebene ' # [rekebene] # ', Ergebnis: ' # [ergebnis]

  ret [ergebnis]
endfunc

rem Erstaufruf der Funktion fac
fac [fakultät] = '4'

rem Ausgabe des Ergebnisses
echo 'Die Fakultät von 4 ist ' # [fakultät]
end
```

Dieses Beispiel zeigt auch nochmal die Verwendung von lokalen und globalen Variablen: Bei jedem rekursiven Aufruf werden alle lokalen Variablen der Prozedur/Funktion neu deklariert und können auch mit neuen bzw. unterschiedlichen Werten gefüllt werden. Es können daher (als Beispiel) mehrere Tausend Variablen [x] im Speicher existieren, doch nur eine ist lokal für den aktuellen Aufruf der Prozedur gültig; Daher können auch nicht die Werte von darunterliegenden Rekursionen geändert werden, wäre dies möglich, wäre rekursive Programmierung völlig unnütz.

Anders verhält es sich bei den altbekannten, globalen Variablen: Diese behalten ihren Wert, so wie unsere Variable [rekebene]. Wir können [rekebene] daher bei jedem Aufruf um 1 inkrementieren (erhöhen) und vor jeder Beendigung der Funktion wieder um eins dekrementieren (verringern). Somit können wir im Meldungsfenster immer genau sehen, in welcher Rekursions-Tiefe wir uns gerade befinden. Wie Sie bei der Ausführung des Scripts sicherlich bemerkt haben, ruft sich die Funktion bei Übergabe des Wertes '4' genau 5 mal wieder selber auf, also bis der Übergabe-Wert 0 ist.

Rekursive Programmierung lässt sich sehr vielfältig einsetzen und kann äusserst effektiv sein.

Um Beispielsweise ein Verzeichnis zu duplizieren wurde in die RapidBATCH-Systemfunktions-Bibliothek (SYSTEM.RB) die Prozedur "copydir" implementiert, die ein komplettes Verzeichnis mitsamt Unterverzeichnissen rekursiv kopieren kann.

```
proc copyDir: [source], [target]
    dec [tok], [i], [files]

    findfiles [files] = [source] # '\*', '1'
    if [files] = ''
        [errorcode] = '-1'
        ret
    endif
    mkdir [target]
    repeat
        calcvar [i] = [i] + '1'
        gettok [tok] = [files], '|', [i]
        if [tok] != ''
            getfileatt [isDir] = [source] # '\' # [tok], 'DIRECTORY'
            if [isDir] = '0'
                copyDir [source] # '\' # [tok], [target] # '\' # [tok]
            else
                copyfile [source] # '\' # [tok], [target] # '\' # [tok]
            endif
        endif
    until [tok] = ''
    [errorcode] = '0'
endproc
```

## Zeiger auf Variablen

Bei unseren bisherigen Beispielen mit Prozeduren haben wir schon oft mit Parametern gearbeitet. Dabei haben wir bisher nur das "Call-By-Value"-Verfahren angewandt, d.h. einen einzelnen Wert an eine lokale Variable der Prozedur übergeben. Bei diesem Verfahren kann aber nur intern innerhalb der aufgerufenen Prozedur mit diesen Werten gearbeitet werden, d.h. sie sind nicht veränderbar, ausserdem lassen sich somit keine Arrays an Prozeduren übergeben. Zu diesem Zwecke gibt es das "Call-by-Reference"-Verfahren, bei dem man einer Prozedur als Parameter nicht den Inhalt der Variablen übergibt, sondern eine lokale "Pseudo-Variable", die dann auf die eigentliche Variable zeigt. Diese "Pseudo-Variablen" heißen Pointer, weil sie auf einen Speicherbereich, eben die Ziel-Variable, zeigen. Über solch einen Pointer kann man auf Variablen (und natürlich auch auf Arrays) uneingeschränkt zugreifen, d.h. Werte lesen und schreiben. Obwohl der Pointer eine lokale Variable ist, kann man über diesen globale Variablen oder auch die lokalen Variablen einer Prozedur verändern, welche die Prozedur mit dem Pointer aufgerufen hat. Es kommt ganz darauf an, worauf der Pointer zeigt.

Ein Parameter wird bei der Prozedurdeklaration mit einem Sternchen (\*) beginnend als Pointer deklariert. Folgendes Beispiel-Script zeigt, wie man einen Pointer in einer Prozedur deklariert, übergibt und verwendet:

```
proc neuer_wert: [*a]
    echo 'Der Wert von [a] ist ' # [a]
    [a] = 'Das ist der neue Wert, der über den Pointer geschrieben wurde'
    echo 'Der Wert von [a] ist jetzt ' # [a]
endproc
```



```
[test] = 'Hello World'
echo 'Der Wert von [test] ist ' # [test]
neuer_wert [test]
echo 'Der Wert von [test] ist jetzt ' # [test]
end
```

Pointer sind auch die einzige Möglichkeit, ein Array in einer Prozedur zu verarbeiten. Bestes Beispiel ist die Prozedur "viewArray" aus der RapidBATCH-Bibliothek ARRAY.RB.

```
rem Prozedur viewArray zur Anzeige eines Arrays
proc viewArray: [*array]
  dec [i], [text]

  if [array:'0'] = '0' ret
  repeat
    [i] + '1'
    [text] # 'Index ' # [i] # ': ' # [array[i]] # [new_line]
  until [i] = [array:'0']

  echo [text]
endproc

rem Array erzeugen
[namen:'1'] = 'Jan'
[namen:'2'] = 'Anna'
[namen:'3'] = 'Stefan'
[namen:'4'] = 'Matthias'
[namen:'5'] = 'Sebastian'
[namen:'6'] = 'Daniel'
[namen:'7'] = 'Uli'
[namen:'8'] = 'Andreas'
[namen:'9'] = 'Volker'
[namen:'10'] = 'Marco'

rem Array anzeigen
viewarray [namen]
end
```

Natürlich können auch die Werte eines Arrays beliebig verändert werden. Folgendes Script quadriert jeden Eintrag eines Arrays und schreibt ihn in das Array selbst zurück.

```
rem Prozedur viewArray zur Anzeige eines Arrays
proc viewArray: [*array]
  dec [i], [text]

  if [array:'0'] = '0' ret
  repeat
    [i] + '1'
    [text] # 'Index ' # [i] # ': ' # [array[i]] # [new_line]
  until [i] = [array:'0']

  echo [text]
endproc

rem Prozedur array_zum_quadrat quadriert jeden Eintrag eines Arrays
proc array_zum_quadrat: [*array]
  dec [i], [text]

  if [array:'0'] = '0' ret
  repeat
    [i] + '1'
    [array:[i]] * [array:[i]]
  until [i] = [array:'0']
endproc

rem Array erzeugen
[zahlen:'1'] = '5'
```

```
[zahlen:'2'] = '7'  
[zahlen:'3'] = '2'  
[zahlen:'4'] = '3'  
[zahlen:'5'] = '9'
```

```
rem Werte ausgeben  
viewarray [zahlen]
```

```
rem Alle Werte im Array quadrieren  
array_zum_quadrat [zahlen]
```

```
rem Werte ausgeben  
viewarray [zahlen]  
end
```

Ein Pointer kann auch als Pointer an eine weitere Prozedur übergeben werden. In diesem Fall würde RapidBATCH über einen Pointer auf einen Pointer auf eine Variable zugreifen! Theoretisch könnte solch eine Kette von Pointern bis ins Unendliche fortgeführt werden.

## Build-in Dialogfenster

### Meldungen und Statusausgaben

RapidBATCH ist mit einer Vielzahl von build-in Dialogen ausgestattet, die individuell angepasst und optimal in Ihren Scripts eingesetzt werden können. Bevor Sie anfangen, selber komplett eigene Dialoge in RapidBATCH zu kreieren (was auch möglich ist) sollten Sie erst einmal schauen, ob bereits die RapidBATCH build-in Dialoge für Ihren Anwendungsfall ausreichen. Der Vorteil bei den build-in Dialogen von RapidBATCH besteht nämlich darin, dass Sie nur eine einzige Anweisung oder Funktion aufrufen müssen, um den kompletten Dialog zu starten. Bei selbst programmierten Dialogfenstern, wozu wir in einem späteren Kapitel kommen werden, müssen Sie für jedes Ereignis mit oder auf dem Dialog eine entsprechende Routine programmieren, was dann wiederum mehr Aufwand erfordert. Dieses Kapitel stellt nun die von RapidBATCH bereitgestellten build-in Dialoge vor und zeigt auf, wie diese angewendet und angepasst werden können.

Meldungsfenster, so genannte Messageboxes, bieten die einfachste Art, dem Benutzer Meldungen oder Daten zu präsentieren oder auch um einfache Abfragen bereitzustellen. Bereits kennengelernt haben Sie die ECHO-Anweisung, mit der Sie ein Meldungsfenster mit dem übergebenen Wert auf dem Bildschirm ausgeben lassen können. ECHO dient zur einfachsten Ausgabe einer Information, daher wird sie auch oft in unseren Beispielen benutzt.



Einfache Ausgabe von Informationen und Meldungen mit dem ECHO-Dialog

Ähnlichen Zweck erfüllt auch die CONFIRM-Funktion, die einen Benutzerdialog mit Ja-Nein-Option anzeigt. CONFIRM erwartet als Parameter eine Rückgabe-Variable (die Auskunft über den gedrückten Button gibt) und einen Aufforderungstext bzw. eine Frage, die mit "Ja" oder "Nein" beantwortet werden kann. Wenn der Button "Ja" geklickt wurde, wird der Wert '0' an die Rückgabe-Variable zurückgegeben, bei "Nein" der Wert '-1'.



Ja-Nein-Auswahl mit dem CONFIRM-Dialog

Ein Beispiel zur Verwendung von CONFIRM ist folgendes Script:

```
%start
rem Abfrage: Script beenden?
confirm [ende] = 'Möchten Sie das Script beenden?'

rem Wenn "Nein" gedrückt wurde, springe zum Anfang
if [ende] = '-1' goto start
echo 'Script beendet.'
end
```

ECHO und CONFIRM bieten somit eine einfache Möglichkeit, schnell einen Wert zu präsentieren oder eine einfache Ja-Nein-Abfrage zu gestalten.

Um aber eine MessageBox individueller zu gestalten, bietet Ihnen RapidBATCH die MSGBOX-Anweisung. MSGBOX benötigt als Parameter einen Titel, einen Meldungstext sowie einen MessageBox-Style, der als Zahl definiert wird. Eine ganz einfache MessageBox wäre hier:

```
msgbox 'Meldung', 'Hello World', '0'
```


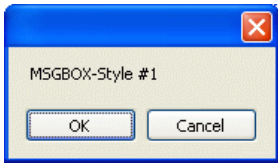
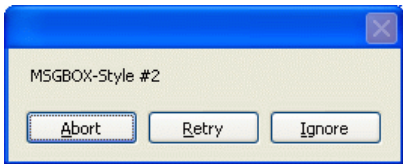
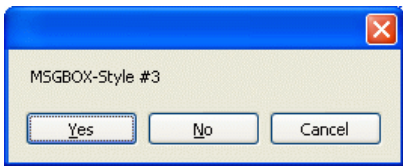
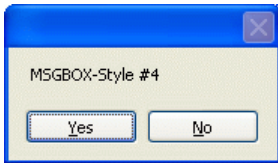
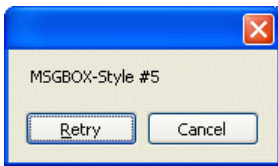
Welcher der angezeigten Buttons gedrückt wurde, wird in die vordeklarierte Variable [errorcode] gespeichert (im obigen Beispiel immer 0, weil ja nur ein Button zur Auswahl steht). [errorcode] ist eine Variable, die von vielen Anweisungen in RapidBATCH verschiedene Status- oder Fehlercodes erhält. Eben hier, welcher Button gedrückt wurde.

Anmerkung: Die MSGBOX-Anweisung kann auch als Funktion aufgerufen werden! Der Wert, der dann an [errorcode] übergeben wird, wird stattdessen bei dieser Methode in die entsprechende Rückgabe-Variable gespeichert. MSGBOX






sollte daher vermehrt als Funktion eingesetzt werden, kann aber natürlich auch weiterhin als Anweisung, wie hier beschrieben, aufgerufen werden.

Ein MessageBox-Style setzt sich aus einem so genannten Button-Wert und einem Icon-Wert zusammen. Zudem lässt sich auch die Position des Eingabefokus (welcher Button gedrückt wird, wenn der Benutzer nur die Entertaste betätigt) mit diesem Wert bestimmen.

Mögliche Button-Styles für Messageboxes sind:

Style-ID:	Beispiel	Button-Style (jew. Rückgabewert an [errorcode] bzw. Return-variable in Klammern):
0		OK (1)
1		OK (1) + CANCEL (2)
2		CANCEL (3) + RETRY (4) + IGNORE (5)
3		YES (6) + NO (7) + CANCEL (2)
4		YES (6) + NO (7)
5		RETRY (4) + CANCEL (2)

Rechnet man zum gewünschten Button-Style noch einen der folgenden Werte hinzu, so kann man auch eines der vier Standard-Icons bestimmen.

Icon-Style:	Beispiel	Icon-Style:	Beispiel	Icon-Style:	Beispiel
0		16		32	
48		64			

Ein Beispiel wäre hier

```
%start
msgbox 'Eine Frage!', 'Klicken Sie einen der Buttons!', '48' + '4'

if [errorcode] = '6'
    rem Fall: Ja
    echo 'Sie haben "Ja" angeklickt'
    goto start
elseif [errorcode] = '7'
    rem Fall: Nein
    echo 'Sie haben "Nein" angeklickt'
    goto start
else
    rem Fall: Abbruch
    echo 'Und tschüss...'
endif
end
```

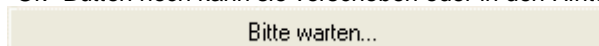
Wenn man zum Style noch den Wert 0 (für den ersten), 256 (für den zweiten) oder 512 (für den dritten) hinzurechnet, kann man den Eingabefokus auch direkt auf einen der Buttons setzen. Welcher Button den Focus hat sieht man an dem kleinen, getrickelten Kasten um den Button herum. Hat ein Button den Focus, so wird dieser Button ausgelöst, wenn der Benutzer einfach die Eingabetaste betätigt.

Beispiel:

```
msgbox 'Test1', 'Fokus 1', '64' + '3' + '0'
msgbox 'Test2', 'Fokus 2', '64' + '3' + '256'
msgbox 'Test3', 'Fokus 3', '64' + '3' + '512'
end
```

Neben den bis jetzt besprochenen Messageboxes bietet RapidBATCH aber auch eine andere Form der Meldungs- und Statusübermittlung: Die INFOBOX!

Manchmal ist es nötig, z.B. bei länger andauernden Operationen, eine Meldung in der Form "Bitte warten..." einzublenden. Die INFOBOX kann vom Programmierer daher völlig frei ein- oder ausgeblendet werden und besitzt weder einen "Ok"-Button noch kann sie verschoben oder in den Hintergrund gebracht werden.



Einblendung des INFOBOX-Dialoges

Ein einfaches Beispiel für die Verwendung der INFOBOX ist folgendes Script, das eine INFOBOX mit dem Text "Hello World" für 5 Sekunden einblendet. Zum Abwarten der 5 Sekunden wird hier das Script mit Hilfe der WAIT-Anweisung für 5000 Millisekunden (= 5 Sekunden) angehalten.

```
infobox 'Hello World', 'show'
wait '5000'
end
```

In unserem Fall wird die INFOBOX automatisch wieder ausgeblendet, weil das Script beendet ist. Um die INFOBOX manuell auszublenden, benutzen Sie INFOBOX noch einmal in der Form

```
infobox '', 'hide'
```

oder verwenden sie die professionellere Methode, indem Sie die vordeklarierte Variable [InfoBox\_Enabled] auf den Wert 'hide' oder '-1' setzen (durch setzen auf 'show' oder '0' wird INFOBOX wieder angezeigt).

Beispiel:

```
echo 'INFOBOX wird jetzt eingeblendet...'
infobox 'Hello World', 'show'
wait '2000'
echo 'INFOBOX wird jetzt ausgeblendet...'
[InfoBox_Enabled] = 'hide'
echo 'INFOBOX ist jetzt ausgeblendet!'
end
```

Da INFOBOX ein komplett eigener Dialog von RapidBATCH ist (Messageboxes werden von Windows selber bereitgestellt), lässt er sich auch individualisieren, z.B. kann die Höhe und Breite des Dialogs mit Hilfe der vordeklarierten Variablen [InfoBox\_Height] und [InfoBox\_Width] vom Programmierer beeinflusst werden. Die Angabe der Größe erfolgt in Pixeln, die Zentrierung des Dialogs auf dem Bildschirm findet automatisch statt, kann aber auch durch das Setzen der Variablen [InfoBox\_X] und [InfoBox\_Y] auf eine beliebige Position des Bildschirms erfolgen. Wird [InfoBox\_X] und/oder [InfoBox\_Y] auf den Wert '0' gesetzt, erfolgt eine automatische Zentrierung des Dialogs auf der jeweiligen Achse (was der Standard-Einstellung entspricht).

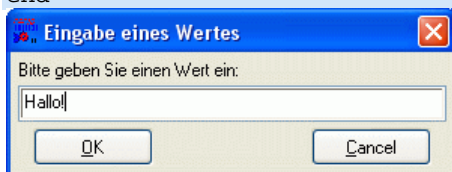
## Abfragen von Werten

Um Werte vom Benutzer abzufragen, stellt RapidBATCH gleich drei verschiedene, leistungsstarke Dialoge zur Verfügung.

Häufigste Verwendung findet dabei die bereits kennengelernte INPUTBOX. Dieser Dialog ermöglicht eine Eingabeaufforderung, in die der Benutzer einen Wert, z.B. einen Namen oder eine Zahl eingeben kann. INPUTBOX ist eine Funktion, deren Rückgabewert der eingegebene String ist, und benötigt drei Parameter, nämlich den Dialog-Titel, einen Aufforderungstext sowie einen Vorgabewert, der direkt im Eingabefeld eingeblendet wird und vom Script-Benutzer geändert oder auch ganz ersetzt werden kann.

Ein einfaches Beispiel ist das bereits im Grundlagenkapitel benutzte Script, bei dem der Benutzer einen Wert eingeben kann, der dann in einer ECHO-Messagebox wieder ausgegeben wird:

```
inputbox [wert] = 'Eingabe eines Wertes', 'Bitte geben Sie einen Wert ein:', 'Hallo!'
echo 'Der eingegebene Wert lautet: ' # [wert]
end
```



Der INPUTBOX-Dialog zur Abfrage von beliebigen Werten

Klickt der Benutzer auf den OK-Button im Dialog (oder auch einfach nur Return), wird der in das Textfeld eingegebene Text zurückgegeben. Klickt der Benutzer den Cancel-Button oder den Schließen-Button des Fensters rechts oben, liefert die Funktion einen Leerstring zurück.

Die Beschriftung der Buttons lässt sich durch Setzen der vordeklarierten Variablen [InputBox\_Ok] und [InputBox\_Cancel] verändern, so daß z.B. auch deutsche Bezeichnungen für die Buttons dort angegeben werden können. Defaultmäßig wird die INPUTBOX zentriert auf dem Bildschirm dargestellt; Eine beliebige, individuelle Anzeigeposition des Dialogs lässt sich allerdings mit Hilfe der vordeklarierten Variablen [InputBox\_X] und [InputBox\_Y] erzielen. Setzt man diese auf den Wert '0' (Standardwert), so bewirkt dies die Zentrierung des Dialogs auf der jeweiligen Achse.

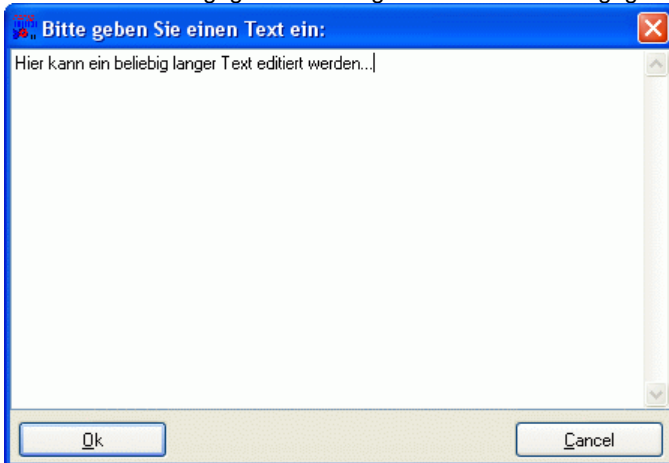
```
[InputBox_Ok] = 'Alles klar!'
[InputBox_Cancel] = 'Stop!'
inputbox [test] = 'Geänderte Buttons', 'Bitte geben Sie etwas ein:', 'hello world'
echo [test]
end
```

Neben dem INPUTBOX-Dialog bietet RapidBATCH auch den PWDBOX-Dialog. Eine PWDBOX ist nichts anderes als eine "INPUTBOX für Passwörter", die zur Eingabe des Wertes anstatt eines normalen Textfeldes ein maskiertes Eingabefeld zur Verfügung stellt (in dem jedes eingegebene Zeichen durch ein "\*" maskiert wird). Ein PWDBOX-Dialog kann (und sollte!) daher auch nur zur verdeckten Eingabe von Passwörtern eingesetzt werden (daher auch der Name: "PWD" ist die englische Abkürzung für "Password"). Die Parameter von PWDBOX sind dieselben wie bei der INPUTBOX (mit selber Funktion), zudem lassen sich, da PWDBOX das selbe Fenster verwendet wie INPUTBOX, die Beschriftung der Buttons ebenfalls über [InputBox\_Ok] und [InputBox\_Cancel] ändern (ebenso [InputBox\_X] und [InputBox\_Y] für die Positionierung des Dialogs auf dem Bildschirm).

```
pwdbox [passwort] = 'Passwort', 'Bitte geben Sie Ihr Passwort ein:', ''
echo 'Ihr Passwort lautet: ' # [passwort]
end
```

Bisher haben wir nur einzeilige Eingaben tätigen können; Diese reichen natürlich für sehr viele Anwendungsbereiche bereits aus. Wenn wir aber nun größere Eingaben, z.B. eine Adresse oder sogar einen ganzen Text vom Script-User abfragen möchten, genügen Dialoge wie INPUTBOX oder PWDBOX nicht mehr.

Hier müssen wir auf ein weiteres Dialogfenster von RapidBATCH zugreifen: Die EDITBOX! EDITBOX ermöglicht es, einen kompletten Text vom Benutzer editieren zu lassen. Ähnlich wie in einem Text-Editor lässt sich in der EDITBOX das Textfenster scrollen und durch Ziehen mit der Maus vergrößern oder verkleinern. EDITBOX arbeitet im Grunde genauso wie INPUTBOX, da durch Klicken des am unteren Rand befindlichen OK-Buttons der im Textfeld eingegebene/editierte Text, oder durch anklicken des Cancel-Buttons bzw. des Schließen-Buttons rechts oben, ein Leerstring an die beim Funktionsaufruf angegebene Rückgabe-Variable zurückgegeben wird.



Der EDITBOX-Dialog zur mehrzeiligen Eingabe bzw. Bearbeitung von Texten und Werten

Als Parameter erwartet die EDITBOX-Funktion einen Fenstertitel, den Text, der im Editierfeld stehen soll, und einen Style-Parameter, ob der Text im Editierfeld editierbar (Wert 'writeable' oder '0') oder nicht-editierbar (Wert 'readonly' oder '-1') ist. Dieser Parameter ermöglicht dann auch die Anzeige von Text ohne dass der Benutzer diesen editieren kann (beispielsweise um eine ReadMe-Datei oder einen Lizenztext bei einem Setup-Script anzuzeigen).

Beispiel:

```
rem Texte editieren mit EDITBOX
editbox [text] = 'Bitte geben Sie einen Text ein:', '', 'writeable'
msgbox 'Der Text lautet:', [text], '64'
end
```

Der EDITBOX-Dialog kann mit Hilfe der vordefinierten Variablen [EditBox\_Width] und [EditBox\_Height] in der Größe angepasst werden, ebenso ist auch die Positionierung des Dialogs auf dem Bildschirm mit den Variablen [EditBox\_X] und [EditBox\_Y] ohne weiteres möglich, wobei auch hier eine automatische (und bereits vordefinierte) Zentrierung des Dialogs mit dem Wert 0 ermöglicht wird.

Die Beschriftung der Buttons ist über [EditBox\_Ok] und [EditBox\_Cancel] möglich. Wenn man die Button-Beschriftung auf einen Leerstring setzt, wird der entsprechende Button einfach ausgeblendet, so dass man z.B. eine EDITBOX mit nur einem "Weiter"-Button programmieren kann, wie in folgendem Beispiel (die Funktionen OPENBOX und READFILE werden später noch genauer besprochen!):

```
rem Datei-Anzeige Script mit der EDITBOX
rem Dateiname abfragen
openbox [file] = 'Bitte Datei wählen:', 'Textdateien|*.txt'
if [file] = '' halt
```



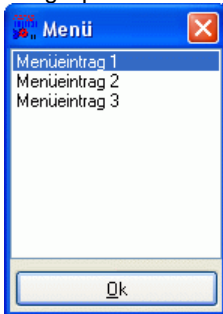
```
rem Datei lesen
readfile [text] = [file], '0'

rem EDITBOX individualisieren und anzeigen
[EditBox_Ok] = ''
[EditBox_Cancel] = 'Weiter >'
[EditBox_Width] = '450'
[EditBox_Height] = '480'
editbox [text] = 'Inhalt von: ' # [file], [text], 'readonly'
```

## Menüs und Auswahlmöglichkeiten

Bisher haben Sie bereits einige Möglichkeiten kennengelernt, um Daten vom Benutzer einzulesen. Diese Daten sind jedoch völlig frei vom Benutzer definierbar. Um aber dem Benutzer die Auswahl eines Wertes aus einer bestimmten Anzahl von Werten zu ermöglichen, stellt RapidBATCH auch zwei leistungsstarke Dialoge zur Verfügung.

Die LISTBOX findet hierbei am häufigsten Verwendung. LISTBOX ist ein Dialog mit einer Listenauswahlmöglichkeit, die sehr vielseitig eingesetzt werden kann: Ob als Programm-Menü, Dateiauswahl oder für andere Zwecke - Das Anwendungsspektrum für den LISTBOX-Dialog ist äusserst breit und vielseitig gehalten.



Der LISTBOX-Dialog bietet eine vielseitige und schnell realisierte Auswahlmöglichkeit

LISTBOX benötigt zur Anwendung zwei Parameter. Einmal einen Dialog-Titel und eine so genannte Liste. Solch eine Liste ist ein String, dessen einzelne "Elemente", die später in der LISTBOX angezeigt werden, standardmäßig durch Pipe-Zeichen (|) getrennt sind (dieses Listentrennzeichen lässt sich global über die Variable [std\_sep] auch ändern). Ein Array lässt sich als Liste nicht direkt übergeben, kann aber mit Hilfe der Funktion "arrayToList" aus der ARRAY.RB-Funktionsbibliothek schnell und einfach in eine Item-Liste für den LISTBOX-Dialog umgewandelt werden.

Rückgabewert der LISTBOX-Funktion ist der ausgewählte Wert aus der Liste. Ist dieser Wert leer, so wurde entweder kein Menüeintrag ausgewählt oder der Schließen-Button der LISTBOX betätigt.

Ein einfaches Beispiel für eine LISTBOX ist folgendes Script mit einer kleinen Auswahl:

```
rem Einfache Auswahl mit einer LISTBOX:
listbox [auswahl] = 'Menü', 'Menüeintrag 1|Menüeintrag 2|Menüeintrag 3'

if [auswahl] = ''
    echo 'Kein Element ausgewählt!'
else
    echo 'Gewähltes Element ist ' # [auswahl]
endif
end
```

Die LISTBOX ist ebenfalls sehr anpassungsfähig. Äquivalent zu den bisher vorgestellten Dialogen lässt sich auch die LISTBOX mit Hilfe der Variablen [ListBox\_Width] und [ListBox\_Height] in der Größe verändern; [ListBox\_X] und [ListBox\_Y] ermöglichen die individuelle Positionierung des Dialoges (Defaultwert ist auch hier '0' zur automatischen Zentrierung). Mit [ListBox\_Ok] lässt sich auch der Text auf dem OK-Button der LISTBOX verändern.

Um die Auswahl von mehr als einem Eintrag aus einer LISTBOX zu ermöglichen, bietet RapidBATCH ausserdem die Variable [ListBox\_MultiSel]. Diese ist standardmäßig auf den Wert '-1' gesetzt, was soviel bedeutet wie "Keine Mehrfachauswahl". Setzt man den Wert auf '0', lassen sich in der LISTBOX mehr als ein Eintrag auswählen, wenn nötig auch alle. In diesem Fall wird ebenfalls (im Fall von mehreren selektierten Einträgen) eine Liste von durch Pipe-Zeichen (bzw. das durch die Variable [std\_sep] definierte Zeichen) von einander getrennten Werten zurückgeliefert.

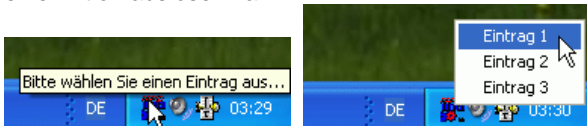
Folgendes Beispiel demonstriert die Verwendung der o.g. Variablen. Beobachten Sie bitte auch den Rückgabewert von LISTBOX bei einer Mehrfach-Auswahl.

```
rem Dateiliste generieren (Dateinamen werden von FINDFILES als Liste zurückgegeben)
findfiles [dateiliste] = 'C:\WINDOWS\*.*', '1'

rem Dateiauswahl ermöglichen mit LISTBOX
[ListBox_Width] = '300'
[ListBox_Height] = '500'
[ListBox_MultiSel] = '0'
[ListBox_Ok] = 'Auswählen!'
listbox [auswahl] = 'Bitte Datei wählen:', [dateiliste]

if [auswahl] = ''
    echo 'Kein Element ausgewählt!'
else
    echo 'Gewählte Datei(en): ' # [auswahl]
endif
end
```

Neben der LISTBOX bietet RapidBATCH seit der Version 5.0 nun auch eine neue, professionelle Menü-Auswahlmöglichkeit: Den TRAYMENU-Dialog! TRAYMENU ermöglicht das Anzeigen eines so genannten Traybar-Menüs in der Windows-Taskleiste. Der "Dialog" selbst wird durch ein kleines Icon repräsentiert, wobei man das Menü mit einem Rechtsklick auf das Icon anzeigen kann. Mit dieser Möglichkeit kann man beispielsweise ein Script entwickeln, welches permanent im Hintergrund läuft, aber mittels Auswahl von Programmfunktionen oder durch Klicken des Icons eine Aktion auslösen kann.



Einfache aber effektive Auswahl- und Menüprogrammierung mit dem TRAYMENU-Menüdialog

Im Grunde genommen wird TRAYMENU genauso verwendet wie die LISTBOX. Man übergibt einen Titel (der als Tooltip angezeigt wird, wenn der Benutzer mit der Maus über das Icon fährt) und eine Liste an Menü-Einträgen, getrennt durch Pipe-Zeichen (diese kann auch leer sein, dann wird gar kein Menü erzeugt). Klickt der Benutzer nun auf das Icon des Tray-Menüs, so liefert die Funktion einen Leerstring zurück; Andernfalls, wenn der Benutzer einen Eintrag aus dem Menü auswählt, wird der entsprechende Menü-String zurückgeliefert.

```
traymenu [eintrag] = 'Bitte wählen Sie einen Eintrag aus...', 'Eintrag 1|Eintrag 2|Eintrag 3'

if [eintrag] ! ''
    echo 'Der gewählte Menüpunkt lautet: ' # [eintrag]
else
    echo 'Das Icon wurde angeklickt.'
endif
end
```

Um ein Menü professionell mit Shortcuts auszustatten, kann man auch einen Buchstaben jedes Eintrags mit einem &-Zeichen versehen. Dies weist RapidBATCH an, den nachfolgenden Buchstaben zu unterstreichen, so dass dieser als Shortcut fungiert (wenn das Menü nun offen ist lässt sich mit diesem Shortcut ein Eintrag durch Tastatureingabe auswählen). Beispiel:

```
traymenu [eintrag] = 'Bitte wählen Sie einen Eintrag aus...', 'Eintrag &1|Eintrag &2|Eintrag &3'
```

Jetzt kann man die einzelnen Einträge mit 1, 2 oder 3 auch über die Tastatur auswählen. Um ein &-Zeichen im Menü auszugeben schreibt man einfach stattdessen "&&", z.B.

```
traymenu [test] = 'Ein Test', 'Hello && World'
```

Übrigens: das &-Zeichen wird, wenn ein Eintrag ausgewählt wurde, wieder mit übergeben, daher bitte auch in Bedingungen auf den Returnwert von TRAYMENU abprüfen!

Um eine Trennlinie in das Menü einzubauen (z.B. um viele Menüeinträge in kleine Gruppen zu trennen), schreibt man einfach einen einzelnen Bindestrich als Menüeintrag. Beispiel:

```
traymenu [eintrag] = 'Bitte wählen Sie einen Eintrag aus...', 'Eintrag &1|Eintrag &2|Eintrag &3|-|B&eenden'
```

TRAYMENU verwendet als Icon immer das Standard-Icon der Applikation. Bei interpretierten Scripts ist dies immer das Icon des RapidBATCH-Interpreters, bei compilierten Scripts das entsprechend hinzugelinkte Programmicon.

Mit Hilfe der vordeklarierten Variablen [app\_icon] lässt sich allerdings auch eine externe ICO-Datei als Icon benutzen. [app\_icon] setzt dabei ein Icon, das für alle Fenster der Anwendung gilt, d.h. es lässt sich hierüber auch das Icon, welches bei LISTBOX, INFOBOX, EDITBOX und INPUTBOX links oben erscheint, mit ändern. Wird [app\_icon] auf einen Leerstring gesetzt, wird automatisch das Standard-Icon der Anwendung wieder benutzt.

Beispiel:

```
rem TRAYMENU unter Verwendung eines individuellen Programm-Icons
[app_icon] = '..\icons\orca1.ico'
traymenu [eintrag] = 'Bitte wählen Sie einen Eintrag aus...', 'Eintrag &1|Eintrag
&2|Eintrag &3|-|B&enden'

if [eintrag] ! ''
    echo 'Der gewählte Menüpunkt lautet: ' # [eintrag]
else
    echo 'Das Icon wurde angeklickt.'
endif
end
```

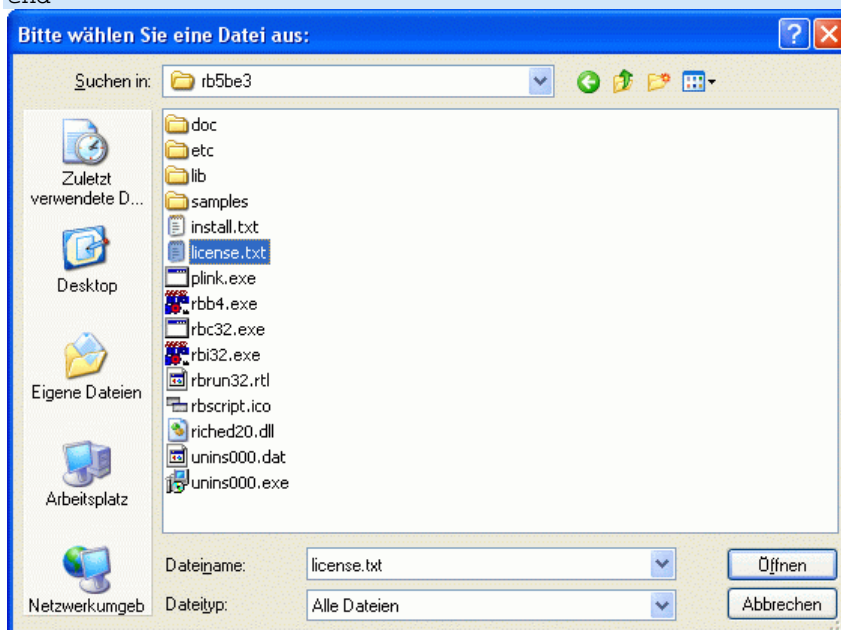
## Datei- und Verzeichnisauswahl

Nicht selten kommt es vor, dass man Datei- oder Verzeichnispfade vom Benutzer abfragen muss. Der Einsatz einer INPUTBOX wäre hier zwar auch möglich, aber unangebracht. Zur Lösung dieses Problems stellt RapidBATCH drei Dialoge zur Verfügung: OPENBOX, SAVEBOX und FOLDERBOX!

Die Dialoge OPENBOX und SAVEBOX ermöglichen die Eingabe eines Dateinamens sowie das Browsen durch Verzeichnisse. Des weiteren kann ein Dateifilter gesetzt werden, der nur bestimmte Dateitypen zur Anzeige bringt. Der einzige Unterschied zwischen OPENBOX und SAVEBOX besteht darin, dass eine OPENBOX einen "Datei öffnen"-Dialog bereitstellt, während eine SAVEBOX einen "Datei speichern"-Dialog anzeigt.

Folgendes Beispiel zeigt die Verwendung der OPENBOX zur Auswahl einer Datei.

```
openbox [datei] = 'Bitte wählen Sie eine Datei aus:', 'Alle Dateien|*.*'
echo 'Der Pfad zur gewählten Datei lautet: ' # [datei]
end
```



Die OPENBOX-Funktion stellt einen "Datei öffnen"-Dialog zur Verfügung

Die OPENBOX-Funktion (alle Parameter sind hierbei identisch für die SAVEBOX) benötigt neben der Rückgabeveriable einen Dialog-Titel (oder einen Aufforderungstext) und den bereits angesprochenen Dateifilter.

Dieser Dateifilter kann sehr individuell gestaltet werden und ermöglicht den Einsatz der Wildcard-Platzhalter \* und ?.

Übergeben wird der Dateifilter als Liste ähnlich wie bei LISTBOX oder TRAYMENU, nur dass hier nach folgendem Muster vorgegangen werden muss:

Filterbezeichnung | Filter

Obiges Beispiel legt als einen Filter \*.\* (alle Dateinamen, die eine Dateiergung haben) an und benennt ihn mit "Alle Dateien". Wenn wir nun alle Textdateien (Dateiergung: TXT) filtern möchten, verwenden wir folgenden Filter:

Textdateien|\*.txt

Die OPENBOX ermöglicht dem Benutzer auch die Auswahl eines Filters, falls mehrere gegeben sind. Man kann also einfach einen weiteren Filter an einen bereits vorhandenen dranhängen, indem man das obige Muster einfach durch ein weiteres Pipe-Listentrennzeichen vom vorhergehenden Filter trennt. Folgendes Beispiel legt drei Filter an.

EXE-Dateien|\*.exe|COM-Dateien|\*.com|Batch-Dateien|\*.bat

Es lassen sich auch mehrere Filtermasken unter einem einzelnen Filternamen speichern. Hierbei trennt man jeden Filter durch ein Semikolon. Folgender Filter ermöglicht die Anzeiger aller Dateien oder nur die Anzeige von ausführbaren Programmdateien (EXE, COM, BAT):

Ausführbare Programme|\*.exe;\*.com;\*.bat|Alle Dateien|\*.\*

Standardmässig lässt sich in einem OPENBOX- bzw. SAVEBOX-Dialog nur eine einzelne Datei anwählen. Eine Mehrfach-Auswahl ist jedoch auch ohne Probleme möglich, indem man für die OPENBOX die Variable [OpenBox\_MultiSel] auf den Wert '0' setzt (äquivalent [SaveBox\_MultiSel] für die SAVEBOX). Ist diese Variable auf dem Wert '0', so lassen sich mehrere Dateinamen aus der Dateiliste auswählen. Bestätigt der Benutzer nun seine Eingabe mit OK, gibt der Dialog eine Liste der Dateinamen- und Pfade, jeweils getrennt durch Pipe-Zeichen, zurück, wobei als erstes Listenelement der absolute Pfad zu den Dateien angegeben wird und in den darauf folgenden Elementen die einzelnen Dateinamen (ohne den Pfad).

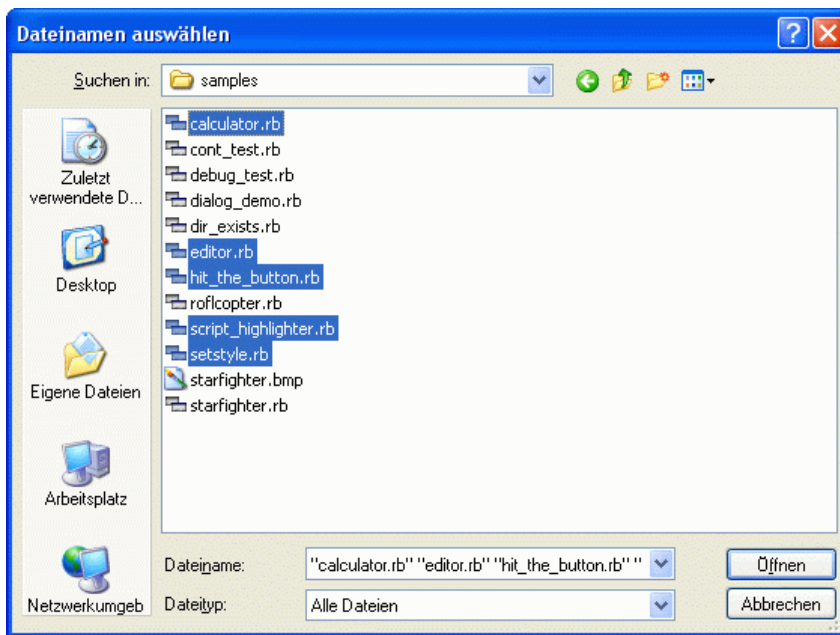
Folgendes Beispiel zeigt, wie man eine Mehrfach-Dateiauswahl verarbeitet.

```
rem Mehrfachauswahl ermöglichen
[OpenBox_MultiSel] = '0'

rem Dateien auswählen
openbox [dateiliste] = 'Dateinamen auswählen', 'Alle Dateien|*.*'

if [dateiliste] != ''
    rem Pfad auslesen
    gettok [pfad] = [dateiliste], '|', '1'

    rem Dateiliste auslesen
    [i] = '1'
    repeat
        [i] + '1'
        gettok [datei] = [dateiliste], '|', [i]
        if [datei] != ''
            echo 'Datei Nr. ' # ([i] - '1') # ': ' # [pfad] # '\' # [datei]
        elseif [datei] = '' & [i] = '2'
            rem Wenn nur eine Datei ausgewählt wurde, steht ihr Name
            rem und Pfad in [pfad]...
            echo 'Einzig gewählte Datei: ' # [pfad]
        endif
    until [datei] = ''
endif
end
```



Mehrfach-Auswahl in einem "Datei öffnen"-Dialog

Wird [OpenBox\_MultiSel] bzw. [SaveBox\_MultiSel] wieder auf den Wert '-1' gesetzt, ist wieder nur eine Einfach-Auswahl mit dem jeweiligen Dialogfenster möglich.

Bei vielen Anwendungen will man auch auf den Komfort nicht verzichten, einfach einen Dateinamen ohne Dateinamenserweiterung in "Datei öffnen"- und "Datei speichern"-Dialog einzugeben. Die Dateinamenserweiterung soll dabei automatisch an den eingegebenen Dateinamen angehängt werden. Um dies zu ermöglichen, kann man eine Standard-Erweiterung für Dateinamen in der Variablen [OpenBox\_DefFileExt] bzw. [SaveBox\_DefFileExt] auf eine beliebige Dateinamenserweiterung (OHNE den Punkt!) setzen:

```
rem Standarderweiterung: TXT-Datei
[SaveBox_DefFileExt] = 'txt'

rem Dateinamen abfragen
savebox [dateiname] = 'Datei speichern als...', 'Textdateien|*.txt'

echo 'Eingegebener Dateiname mit Erweiterung: ' # [dateiname]
end
```



Bequeme Verzeichnisauswahl mit dem "Ordner suchen"-Dialog der FOLDERBOX-Funktion

Neben OPENBOX und SAVEBOX bietet RapidBATCH auch einen Dialog zur Auswahl von Verzeichnissen an: Die FOLDERBOX! Parameter sind hierbei ein Aufforderungstext sowie ein optionaler Basis-Pfad, der automatisch im Dialog angewählt wird.

Folgendes Script zeigt das Windows-Verzeichnis in einem Verzeichnis-Auswahldialog an:

```
rem Verzeichnisauswahl
folderbox [verzeichnis] = 'Bitte wählen Sie ein Verzeichnis:', [windir]

rem Ausgabe des Verzeichnispfades, falls dieses gewählt wurde
if [verzeichnis] ! ''
    echo 'Pfad zum gewählten Verzeichnis: ' # [verzeichnis]
else
    echo 'Sie haben kein Verzeichnis ausgewählt!'
endif
end
```

Wird hier der Abbrechen- oder Schließen-Button des Dialogs gewählt, gibt die Funktion einen Leerstring zurück.

## Systemspezifische Operationen

### Programme starten

Oftmals ist es gerade in der Batch-Programmierung notwendig, externe Programme zu starten, um bestimmte Vorgehensweisen und Methoden, die zu einem Ergebnis führen, zu automatisieren.

Um Programme zu starten, bietet Ihnen RapidBATCH gleich vier verschiedene Möglichkeiten an. Klassische Methode zum Starten einer Anwendung ist die SHELL-Anweisung. Sie führt ein Programm parallel zum RapidBATCH-Script aus, d.h. das Script selbst läuft parallel zum gestarteten Programm weiter. SHELL erwartet als Parameter den Pfad des zu startenden Programms (der auch Kommandozeilen-Parameter beinhalten kann) sowie einen so genannten Anzeigemodus (Show-Mode), der angibt, wie das Programm gestartet werden soll.

Der Anzeigemodus ist nur für einige Windows-Programme interessant, nicht alle Programme unterstützen diese Möglichkeit. Als Anzeigemodi können hier die Werte 'SHOW', 'HIDE', 'MINIMIZED' oder 'MAXIMIZED' angegeben werden; 'SHOW' startet ein Programm mit der normalen Fenstergröße, 'HIDE' startet es versteckt, d.h. nicht sichtbar, womit die 'HIDE'-Option für Konsolenprogramme und Batch-Jobs sehr interessant ist, da diese somit unsichtbar im Hintergrund und ohne ein DOS-Fenster ausgeführt werden können. Die Modi 'MINIMIZED' und 'MAXIMIZED' starten das Programm mit einem minimierten bzw. maximierten Programmfenster. Diese beiden Werte sind jedoch - wie bereits angesprochen - nur bei bestimmten Programmen anwendbar.

Folgendes Script starten den Editor "Notepad" von Windows:

```
shell 'notepad.exe', 'show'
echo 'Und jetzt starten wir Notepad maximiert...'
shell 'notepad.exe', 'maximized'
echo 'Ende!'
end
```

Neben der SHELL-Anweisung bietet RapidBATCH auch die CALL-Anweisung. CALL ist von den Parametern her identisch zu SHELL, hat aber die Eigenschaft, dass das RapidBATCH-Script, welches die Anwendung mit CALL aufgerufen hat, solange mit der Programmausführung wartet, bis das gestartete Programm abgearbeitet oder vom Benutzer beendet wurde.

Hier einmal unser Beispiel von oben, welches aber mit CALL arbeitet:

```
call 'notepad.exe', 'show'
echo 'Und jetzt starten wir Notepad maximiert...'
call 'notepad.exe', 'maximized'
echo 'Ende!'
end
```

Oftmals ist auch die Verwendung von MS-DOS-Konsolenkommandos einfach unabdingbar. Zur schnellen Ausführung eines Konsolenkommandos bietet RapidBATCH die SYSTEM-Anweisung. Als Parameter wird das entsprechende Kommando erwartet.

Beispiel:

```
rem Ping an www.jmksf.de schicken
system 'ping www.jmksf.de'
end
```

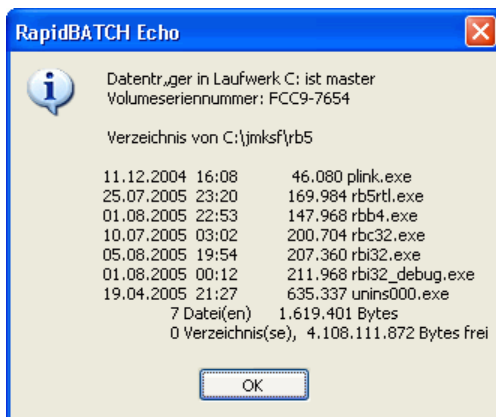
Mit CALL bzw. SHELL lassen sich DOS-Kommandos auch versteckt ausführen (dieses Beispiel verwendet die Funktion READFILE zum Auslesen der Datei, die wir später detaillierter besprechen werden):

```
rem dir-Kommando senden und in Datei test.txt umleiten
call 'cmd.exe /C dir *.exe >test.txt', 'hide'

rem test.txt einlesen und ausgeben
readfile [dir_ausgabe] = 'test.txt', '0'
echo [dir_ausgabe]

rem test.txt wieder löschen
delfile 'test.txt'
end
```





Die Ausgabe des DIR-Befehls, umgeleitet und ausgelesen aus einer Datei

Anmerkung: Unter Windows 95, 98 und ME müssen Sie bei dem oben gezeigten Beispiel anstatt CMD.EXE das Programm COMMAND.COM aufrufen.

Zur letzten Möglichkeit, externe Programme zu starten, gehört die OPEN-Anweisung. OPEN kann vielseitig eingesetzt werden, beispielsweise kann hier auch anstatt eines Programmpfades der Name einer Datei (die dann mit dem entsprechend verknüpften Programm geöffnet wird) oder eine Internet-URL, die dann im Standard-Browser geladen wird, angegeben werden.

Beispiele:

```
open 'calc.exe'
open 'C:\jmksf\rb5\license.txt'
open 'http://www.rapidbatch.com'
end
```

Wenn OPEN zum Starten von Programmen benutzt wird, entspricht dies einem Aufruf von SHELL mit dem Anzeigemodus 'SHOW'.

## Verzeichnisoperationen

Zum Erstellen, Wechseln und Löschen von Verzeichnissen stellt RapidBATCH drei Anweisungen zur Verfügung: MKDIR, CHDIR und RMDIR. Diese sind äquivalent zu den gleichnamigen DOS-Kommandos.

Mit folgender Zeile erzeugen wir ein neues Verzeichnis C:\wale.

```
mkdir 'C:\wale'
```

RapidBATCH legt auch übergeordnete Verzeichnisse mit MKDIR an, wenn wir einen Pfad angeben, der nicht existiert.

Wenn wir also die Zeile

```
mkdir 'C:\wale\zahnwale\schwertwal'
```

ausführen, und weder der Pfad C:\wale noch C:\wale\zahnwale existiert, legt RapidBATCH alle drei Verzeichnisse automatisch hintereinander an.

Möchten wir nun das Verzeichnis wechseln, verwenden wir die CHDIR-Anweisung:

```
chdir 'C:\wale\zahnwale\schwertwal'
```

Ob unser Verzeichniswechsel erfolgreich war oder nicht, können wir anhand der vordeklarierten Variablen [errorcode] überprüfen. Sowohl MKDIR als auch CHDIR und RMDIR geben bei einem Fehler den Wert '-1' (false) an [errorcode], ansonsten den Wert '0' (true) zurück. Wir können also wie folgt herausfinden, ob unser Verzeichnis existiert oder nicht:

```
chdir 'C:\wale\zahnwale\schwertwal'
if [errorcode] = '-1'
    echo 'Verzeichniswechsel fehlgeschlagen!'
else
```

```
echo 'Verzeichniswechsel erfolgreich!'
endif
```

Mit RMDIR löschen wir ein Verzeichnis wieder. Das zu löschende Verzeichnis MUSS allerdings leer sein, darf also weder Dateien noch weitere Unterverzeichnisse enthalten.

```
rmdir 'C:\wale\zahnwale\schwertwal'
```

Anders als bei MKDIR wird hier aber immer nur ein einziges Verzeichnis, nämlich der entsprechend angegebene Pfad gelöscht.

Anmerkung: Verwenden Sie zum Verschieben bzw. Kopieren von Verzeichnissen (samt deren Unterverzeichnissen) die Bibliotheks-Anweisungen MOVEDIR bzw. COPYDIR aus der RapidBATCH-Bibliothek SYSTEM.RB.

## Dateioperationen im Filesystem

Um Dateien zu verwalten, bietet Ihnen RapidBATCH die Anweisungen und Funktionen NEWFILE, DELFILE, COPYFILE, RENAMEFILE, FILEEXISTS und FINDFILES.

Die Anweisung NEWFILE erzeugt eine leere Datei, DELFILE löscht eine existierende Datei. Die Syntax ist bei beiden Anweisungen äquivalent:

```
rem Datei erzeugen
newfile 'leer.txt'
if [errorcode] = '0' echo 'Datei leer.txt wurde erzeugt!'

rem Datei löschen
delfile 'leer.txt'
if [errorcode] = '0' echo 'Datei leer.txt wurde wieder gelöscht!'
```

Auch hier können wir (übrigens bei jeder Datei-spezifischen Anweisung) wieder über die Variable [errorcode] prüfen, ob die jeweilige Dateioperation erfolgreich war ([errorcode] = '0') oder ob nicht ([errorcode] = '-1').

Um eine Datei zu kopieren bietet RapidBATCH die COPYFILE-Anweisung. Sie kopiert den Inhalt im Verhältnis 1:1 aus einer Quelldatei in eine Zieldatei. Die Anweisung RENAMEFILE ermöglicht das Umbenennen bzw. Verschieben einer Datei.

Beispiel:

```
rem C:\jmksf\rb5\readme.txt in C:\test.txt kopieren...
copyfile 'C:\jmksf\rb5\readme.txt', 'C:\test.txt'

rem C:\TEST.TXT öffnen...
open 'C:\test.txt'

rem C:\TEST.TXT wieder löschen...
delfile 'C:\test.txt'

rem C:\jmksf\rb5\readme.txt in C:\test.txt umbenennen...
renamefile 'C:\jmksf\rb5\readme.txt', 'C:\test.txt'
rem C:\TEST.TXT öffnen...
open 'C:\test.txt'

rem C:\test.txt wieder zurück in C:\jmksf\rb5\readme.txt benennen...
renamefile 'C:\test.txt', 'C:\jmksf\rb5\readme.txt'
```

COPYFILE und RENAMEFILE erlauben keine Wildcards, d.h. es kann immer nur eine einzelne Datei kopiert werden. Wie man aber trotzdem mehrere Dateien kopieren/verschieben kann, besprechen wir weiter unten.

Ob eine Datei bereits existiert oder nicht lässt sich mit der Funktion FILEEXISTS überprüfen.

```
fileexists [fx] = 'C:\AnyFile.txt'
if [fx] = '-1'
    echo 'Datei existiert nicht.'
else
```

```
echo 'Datei existiert!'
endif
```

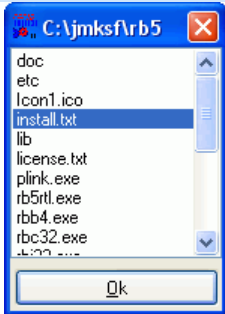
FILEEXISTS gibt also den Wert '-1' an die Rückgabewariable zurück, wenn die Datei nicht existiert. Ansonsten ist der Rückgabewert '0'.

Zu den leistungsstärksten und effektivsten Funktionen, die RapidBATCH bereitstellt, gehört die FINDFILES-Funktion. FINDFILES listet alle Dateien eines Verzeichnisses, ermöglicht aber auch das Einlesen eines Verzeichnisses mitsamt aller Unterverzeichnisse.

Folgendes Script erzeugt eine Liste aller Dateien des aktuellen Verzeichnisses und zeigt diese in einem LISTBOX-Dialog an.

```
rem Dateiliste erzeugen
findfiles [dateiliste] = '*', '1'

rem Anzeige der Dateiliste in einer LISTBOX
listbox [datei] = [current], [dateiliste]
if [datei] != '' echo 'Gewählte Datei: ' # [datei]
end
```



Die mit FINDFILES erzeugte Dateiliste wird in einer LISTBOX zur Anzeige und Auswahl gebracht.

FINDFILES erwartet als Parameter einen so genannten Pfad-/Dateifilter, d.h. es werden nur Dateien, die sich im angegebenen Verzeichnis befinden und dem gegebene Dateifilter entsprechen, gelistet. Für den Dateifilter können so genannte Wildcards benutzt werden; Diese Wildcards (Platzhalter) werden durch die Zeichen \* und ? repräsentiert und ermöglichen das Suchen nach bestimmten Mustern. Ein \* ist ein Wildcard für ein beliebiges Textmuster, ? für ein einzelnes Zeichen.

Der zweite Parameter repräsentiert die Tiefe, bis zu wieviele Unterverzeichnisse eingelesen werden sollen; Der Wert 1 bedeutet, dass nur das aktuelle bzw. angegebene Verzeichnis durchsucht wird, 2 durchsucht das aktuelle bzw. angegebene Verzeichnis und alle darin enthaltenen Unterverzeichnisse. Haben diese Unterverzeichnisse wiederum Unterverzeichnisse, werden diese nicht mehr eingelesen; Wenn man diese auch einlesen will, muss man als Tiefe den Wert 3 angeben. Diese Tiefen-Angaben ließen sich jetzt bis ins Unendliche fortführen. Wird als Tiefe der Wert 0 angegeben, so wird bis zu einer maximalen Verzeichnistiefe von 255 gesucht und gelistet.

Rückgabewert von FINDFILES ist eine Dateiliste, in der jeder Dateiname durch das in der vordefinierten Variablen [std\_sep] definierte Separationszeichen (im Normalfall das Pipe-Zeichen "|") getrennt ist. Wir können diese Liste daher direkt an LISTBOX als Item-Liste übergeben.

Um einmal die Verwendung der verschiedenen Filter, Wildcards und Verzeichnistiefenangaben zu demonstrieren, hier ein paar kommentierte Beispielaufufe von FINDFILES:

```
rem Alle Dateien im aktuellen Verzeichnis auflisten:
findfiles [dateiliste] = '*', '1'
rem (Der Filter "*" sucht nach allen Dateien, die es im Verzeichnis gibt)

rem Alle TXT-Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen
rem bis zu einer Ebene auflisten:
findfiles [dateiliste] = '*.txt', '2'
rem (Der Filter "*.txt" sucht nach allen Dateien, die die Dateierendung
rem .TXT haben. Es werden also z.B. Dateinamen wie "hallo.txt", "x.txt"
rem und "txt.txt" eingelesen (wenn diese denn existieren)).

rem Alle Dateien im aktuellen Verzeichnis auflisten, die mit "hallo"
rem beginnen und mit ".DOC" enden:
findfiles [dateiliste] = 'hallo*.doc', '1'
```

```
rem Alle Dateien im aktuellen Verzeichnis auflisten, die mit "h", einem beliebigen
rem Buchstaben und "llo" beginnen und mit irgend einer Dateiendung enden:
findfiles [dateiliste] = 'h?llo*.*', '1'
rem (Hier werden z.B. Dateinamen wie "hallo92.sav", "hello.bak", "hullokkn.exe"
rem und "hhlloHALLO." gefunden!)
```

```
rem Alle Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen
rem auflisten, die eine Dateiendung haben:
findfiles [dateiliste] = '*.*', '0'
```

```
rem Alle .RB-Dateien im aktuellen Verzeichnis auflisten, die nur einen
rem Buchstaben im Dateinamen haben:
findfiles [dateiliste] = '?rb', '1'
rem (Der Filter "?rb" sucht nach allen Dateien, die einen einzelnen
rem Buchstaben vor dem Punkt haben (? = einzelner Buchstabe), es werden
rem also nur die Dateien "a.rb", "x.rb" oder "l.rb" gelistet (wenn
rem diese denn existieren))
```

Zum Schluss noch ein Beispiel, wie Sie mit Hilfe der Funktionen FINDFILES und GETTOK eine Prozedur programmieren, die mehrere Dateien in ein Zielverzeichnis kopiert (mit Hilfe von COPYFILE, da diese Funktion ja von sich aus keine Wildcards unterstützt):

```
proc copyfiles: [quelle], [ziel], [filter]
    rem Lokale Variablen vereinbaren
    dec [i], [datei], [dateiliste]

    rem Dateiliste holen
    findfiles [dateiliste] = [quelle] # '\' # [filter], '1'
    if [dateiliste] = '' ret

    rem Ziel-Verzeichnis zur Sicherheit erzeugen
    rem (wird natürlich, wenn es schon existiert, nicht überschrieben)
    mkdir [ziel]

    rem Dateiliste durchlaufen
    [i] = '0'
    repeat
        [i] + '1'
        gettok [datei] = [dateiliste], '|', [i]

        rem Wenn [datei] nicht leer ist, Datei von
        rem Quelle nach Ziel kopieren
        if [datei] ! ''
            copyfile [quelle] # '\' # [datei], [ziel] # '\' # [datei]
        endif
    until [datei] = ''
endproc

rem Aufruf (kopiert alle Dateien mit dem Muster "*.*" aus dem
rem Verzeichnis "C:\jmksf\rb5" in das Verzeichnis "C:\test"):
copyfiles 'C:\jmksf\rb5', 'C:\test', '*.*'
end
```

## Dateien lesen und schreiben

Um Dateien in RapidBATCH zu verarbeiten, stellt Ihnen RapidBATCH vier vielseitig anwendbare Anweisungen und Funktionen zur Verfügung. Mit Hilfe dieser Anweisungen und Funktionen können Sie sowohl Textdateien als auch binäre Dateien lesen und schreiben.

Um Daten in eine Textdatei zu schreiben, bietet RapidBATCH die Anweisung WRITEFILE. WRITEFILE erwartet als Parameter einen Dateinamen sowie den zu schreibenden Text (also die zu schreibenden Daten).

Einfachstes Fallbeispiel:

```
writefile 'hello.txt', 'Dies ist ein einfacher Text in einer Textdatei!'
```

WRITEFILE erweitert bei jedem Aufruf den Inhalt einer Datei, falls diese schon existiert, d.h. der zu schreibende Wert wird an das Ende der Datei angehängt. Ein Zeilenumbruch je WRITEFILE-Aufruf geschieht in der Datei automatisch. Existiert die zu schreibende Datei noch nicht, wird sie neu erstellt und der entsprechende Wert in die erste Zeile geschrieben.

Führen wir also den folgenden Code aus:

```
writefile 'test.txt', 'Das ist ein Test!'
writefile 'test.txt', 'Und jetzt sind wir in der zweiten Zeile!'
writefile 'test.txt', 'So denn! Das ist das Ende der Datei.'
```

so erhält die resultierende Datei TEST.TXT folgenden Inhalt:

```
Das ist ein Test!
Und jetzt sind wir in der zweiten Zeile!
So denn! Das ist das Ende der Datei.
```

Wollen wir nun die Datei wieder auslesen, verwenden wir die Funktion READFILE. READFILE ist ebenfalls nur zum Lesen von Textdateien geeignet, da sie eine Datei zeilenweise ausliest. Parameter sind der Dateiname der zu lesenden Datei sowie eine Zeilenangabe. Rückgabewert ist der Inhalt der angeforderten Zeile.

Folgendes Script liest die erste Zeile unserer gerade erstellten TEST.TXT aus.

```
readfile [zeile] = 'test.txt', '1'
echo 'Erste Zeile von TEST.TXT: ' # [zeile]
```

Wollen wir die Datei nun komplett Zeile-für-Zeile lesen, ist ein so genannter Zeilen-Zähler notwendig. Wir verwenden hier die Variable [i]. Wird bei READFILE eine Zeile angefordert, die nicht mehr existiert, so ist der Rückgabewert der Funktion der Wert "EOF", was soviel bedeutet wie "End Of File". Die Abbruchbedingung für unsere Ausleseschleife ist also der Vergleich, ob [zeile] den Wert 'EOF' erhält, womit wir nun folgendes Script erhalten:

```
[i] = '0'
repeat
    rem [i] inkrementieren (erhöhen)
    [i] + '1'

    rem Zeile [i] lesen
    readfile [zeile] = 'test.txt', [i]

    rem Ausgabe der Zeile
    echo 'Inhalt von Zeile ' # [i] # ': ' # [zeile]
until [zeile] = 'EOF'
```

READFILE kann auch eine Datei "in einem Rutsch" einlesen, also die komplette Textdatei zurückgeben. Diesen Effekt erhält man, wenn man als Zeilenindex eine Zahl kleiner oder gleich '0' angibt. Folgendes Script zeigt uns also direkt den Inhalt der Datei TEST.TXT an.

```
readfile [datei] = 'test.txt', '0'
echo 'Inhalt von TEST.TXT:' # [new_line] # [new_line] # [datei]
```

Wir verwenden im obigen Script ausserdem die vordeklarierte Variable [new\_line], um jeweils einen Zeilenumbruch im ECHO-Meldungsfenster zu erzeugen.

Sowohl bei READFILE als auch bei WRITEFILE kann durch Überprüfung der Variablen [errorcode] geprüft werden, ob die jeweilige Aktion erfolgreich war oder nicht. Die Vorgehensweise ist dieselbe wie bei den bisher besprochenen Anwendungsfällen für [errorcode], '-1' bei einem Fehler (z.B. bei READFILE, wenn EOF eintritt oder die Datei nicht existiert, '0' bei erfolgreicher Durchführung).

Neben dem Schreiben und Lesen von Textdateien ist es in RapidBATCH 5 auch erstmals möglich, Binärdateien zu bearbeiten. Als Binärdateien definiert man alle Dateien, die keine direkten Textdateien sind. Selbst Office-Dateien wie .XLS oder .DOC sind KEINE Textdateien.

Wir werden in diesen Beispielen hier auch nur lesbare Texte in die Binärdateien schreiben, es können aber auch beliebige Sonder- und Steuerzeichen in die Binärdateien geschrieben werden.

Um eine Datei (es kann auch eine Textdatei sein!) binär zu schreiben, bietet RapidBATCH die Anweisung PUTFILE. Wenn man mit PUTFILE in eine nicht existierende Datei schreiben will, schlägt dies fehl, daher MUSS die Datei vorher mit NEWFILE explizit erzeugt werden. Folgendes Beispiel schreibt den Text "Hello World" binär in eine Datei BINARY.TXT. Neben dem Dateinamen und dem zu schreibenden Wert muss bei PUTFILE als zweiter Parameter der Byte-

Offset, wo der Text in die Datei eingefügt werden soll, angegeben werden. Als Offset bezeichnet man die Position eines Zeichens ab dem Anfang der Datei. Da unsere Datei leer ist, beginnen wir also bei Offset 1. Es ist also das erste Byte der Datei. Die Zeichen, die binär in die Datei geschrieben werden, werden nicht im Klartext angegeben, sondern als Liste von ASCII-Zeichencodes der zu schreibenden ASCII-Zeichen, in der jeder Wert durch das bereits bekannte Standard-Separatorzeichen, welches über die Variable [std\_sep] definiert werden kann, getrennt ist. Wie bereits bekannt sein sollte ist dies im Normalfall das Pipe-Zeichen "|".

```
newfile 'binary.txt'
putfile 'binary.txt', '1', '72|101|108|108|111|32|87|111|114|108|100'
open 'binary.txt'
```

In diesem Fall haben wir gerade 11 Byte in die Datei eingefügt, also 11 ASCII-Zeichen.

Anmerkung: Um einen Klartext ganz einfach in eine Liste von ASCII-Zeichen zu konvertieren, verwenden Sie einfach folgendes Script, welches ich auch zum Erstellen der Beispiele in diesem Handbuch verwendet habe:

```
inputbox [eingabe] = 'Klartext', 'Bitte geben Sie einen Text ein:', ''
if [eingabe] = '' halt

[i] = '0'
getlen [len] = [eingabe]
[ausgabe] = ''
repeat
    [i] + '1'
    getcharat [ch] = [eingabe], [i]
    getasc [ch] = [ch]
    if [ausgabe] ! '' [ausgabe] # [std_sep]
        [ausgabe] # [ch]
until [i] = [len]

[clipboard] = [ausgabe]
echo 'Der Text wurde als ASCII-Liste dekodiert und in die Zwischenablage kopiert.'
end
```

Wir könnten jetzt auch beispielsweise direkt an Byte-Position 100 mit dem Schreiben der Datei fortfahren:

```
rem Text: "Und dies ist ein weiterer"
[text] = '85|110|100|32|100|97|115|32'
[text] # '|105|115|116|32|101|105|110|32'
[text] # '|119|101|105|116|101|114|101|114'
putfile 'binary.txt', '100', [text]
```

```
rem Text: "Binär-Schreibvorgang ;)"
[text] = '32|66|105|110|228|114|45|83|99'
[text] # '|104|114|101|105|98|118|111'
[text] # '|114|103|97|110|103|32|59|41'
```

```
rem Da wir jetzt 24 Zeichen geschrieben haben,
rem müssen wir ab Pos 100 + 24 weiterschreiben
putfile 'binary.txt', '100' + '24', [text]
```

Diese Zeilen und Mengen an ASCII-Codewerten mögen vielleicht im ersten Moment etwas chaotisch aussehen, da wir hier auf reiner RapidBATCH-Sprachbasis arbeiten. Wenn Sie später die Binärfunktionen produktiv einsetzen wollen ist es natürlich ratsam, sich entsprechende Hilfsfunktionen zu programmieren, die einem die Arbeit erheblich sparen können und sich völlig individuell an die entsprechenden Einsatzumgebungen abändern lassen.

Wenn wir nun zusätzlich auch obige Zeilen ausgeführt haben, sieht die Datei, wenn man sie in einem HEX-Editor betrachtet, so aus:

000000	48 65 6c 6c 6f 20 57 6f 72 6c 64 00 00 00 00 00 00 00 00 00 00 00 00 00	Hello World.....
000014	00 00	.....
000028	00 00	.....
00003c	00 00	.....
000050	00 00	.....
000064	6e 64 20 64 61 73 20 69 73 74 20 65 69 6e 20 77 65 69 74 65 69 6e 20 77 65 69 74 65	nd das ist ein weite
000078	72 65 72 20 42 69 6e e4 72 2d 53 63 68 72 65 69 62 76 6f 72 68 72 65 69 62 76 6f 72	rer Binär-Schreibvor
00008c	67 61 6e 67 20 3b 29 20 3b 29 20 3b 29 20 3b 29 20 3b 29 20 3b 29 20 3b 29 20 3b 29	gang ;)

Inhalt der Datei BINARY.TXT in einem Hex-Editor

Deutlich zu sehen ist, dass unser zweiter, binärer Schreibvorgang bei Hex-Offset 64 beginnt, also Dezimal der Offset 100. Der Bereich zwischen unserem "Hello World" und dem Text "Und das ist ein weiterer Binär-Schreibvorgang ;)" wird mit dem ASCII-Zeichencode 0 aufgefüllt, was soviel heißt wie "nichts" (NUL).

**ACHTUNG:** Bei Windows 95, 98 und ME bzw. FAT und Derivate formatierten Speichermedien wurde beobachtet, dass anstatt ASCII-Zeichencode 0 der Inhalt von gelöschten Daten auf der Festplatte in der Datei auftauchen können, da PUTFILE die Daten ja binär einfach irgendwo auf die Festplatte an die angegebene Position schreibt; Der Grund dafür ist, dass einfach die neu hinzugekommenen Daten zur Datei selbst hinzugefügt werden. Daher bitte, um diesen Effekt zu vermeiden, die entsprechenden Stellen vorher manuell mit NUL- oder einem anderem ASCII-Zeichen auffüllen.

Wenn Sie mit PUTFILE in eine bestehende Datei schreiben, werden alle Zeichen, die Sie an einer bestimmten Byteposition angeben, einfach in die Datei geschrieben, d.h. vorhandene Zeichen werden einfach überschrieben. Es erfolgt also kein append (Anhängen von Daten), denn dies wäre ja für binäre Dateiverarbeitung nicht angebracht, bzw. ohne größeren Sinn (Dateien werden ja automatisch erweitert, wenn Daten über das Dateiende hinaus geschrieben wurden, wie wir es im obigen Beispiel gemacht haben).

Das Auslesen einer Binärdatei funktioniert so ähnlich wie das Schreiben - hierzu benutzen wir gegensätzlich zu PUTFILE die Funktion GETFILE. GETFILE erwartet als Funktionsparameter den Pfad der zu lesenden Datei, den Anfangs-Offset sowie die Anzahl der Bytes, die vom Anfangs-Offset aus gelesen werden. Wenden wir nun GETFILE auf unsere im obigen Beispiel erstellte Datei BINARY.TXT an. Zu beachten sei auch hier, dass die Rückgabe von GETFILE eine Liste an ASCII-Codes ist, die wir zuerst wieder mit der Funktion GETCHR zu entsprechenden Zeichen umwandeln müssen, was aber kein größeres Problem darstellt.

```
rem Daten lesen
getfile [data] = 'binary.txt', '100' + '25', '20'
echo 'Gelesene ASCII-Liste: ' # [data]

rem ASCII-Codes in ASCII-Zeichen umwandeln und an Ausgabestring hängen
[i] = '0'
repeat
    [i] + '1'
    gettok [zeichen] = [data], [std_sep], [i]
    if [zeichen] != ''
        getchr [zeichen] = [zeichen]
        [klartext] # [zeichen]
    endif
until [zeichen] = ''

rem Den Klartext ausgeben
echo 'Klartext: ' # [klartext]
end
```

Wir lesen hier den Text "Binär-Schreibvorgang" aus der oben erstellten und geschriebenen Datei. Wir setzen dabei den Start-Offset auf 125 (dort wo der Text anfängt) und lesen von dort aus 20 Zeichen. Ergebnis ist der entsprechende String!

Wenn Sie mit GETFILE über das Dateiende hinaus lesen wollen, wird entweder ein Leerstring oder aber alle Bytes bis zum Dateiende zurückgegeben.

## Wozu das alles?

Sicherlich haben Sie sich gefragt, warum wir hier einfache Textdateien binär schreiben. Das ist natürlich nicht der eigentliche Sinn und Zweck von Binärdateien. Sie haben aber mit PUTFILE und GETFILE die volle Kontrolle über eine Datei; Bei READFILE und WRITEFILE werden reine text-basierende Dateien verwaltet, doch mit PUTFILE und GETFILE haben Sie einfach mehr Möglichkeiten, eine Datei zu verarbeiten. Sie stellen sozusagen die "professionelle Alternative" zu WRITEFILE und READFILE dar, welche Sie aber mit höchster Wahrscheinlichkeit vermehrt bei der Programmierung von Scripts in RapidBATCH einsetzen werden, da WRITEFILE und READFILE für die meisten Anwendungsfälle mehr als ausreichend und natürlich viel einfacher zu handhaben sind.



## Programme fernsteuern

Eine effektive Möglichkeit der Automatisierung ist auch das Fernsteuern von fremden Anwendungen. RapidBATCH ist dabei in der Lage, anderen Windows-Programmen vorzugaukeln, der PC-Benutzer würde das Programm selbst bedienen, indem es Tastaturbefehle und Tastenkombinationen an das entsprechende Fenster sendet.

Zu diesem Zweck stellt RapidBATCH die SENDKEYS-Anweisung zur Verfügung. Mit einer sehr einfach zu erlernenden und primitiv aufgebauten Makrosprache kann der RapidBATCH-Programmierer über diese Anweisung Tastenkombinationen und Tastatureingaben an theoretisch jedes beliebige Windows-Programm senden.

Folgendes Script demonstriert, wie sich der Windows Editor "Notepad" fernsteuern lässt: Es startet Notepad und gibt den Text "Hello World" ein. Danach wird die Datei gespeichert und das Programm anschliessend beendet, wozu das Script die entsprechenden Tastenkombinationen schickt, um das Menü zu öffnen und den Menüpunkt "Datei -> Speichern" auszuwählen.

```
shell [windir] # '\notepad.exe', 'show'
sendkeys 'Unbenannt - Editor': 'Hello World^(alt)dtest.txt(enter)^(alt)(f4)'
end
```

Als ersten Parameter erwartet SENDKEYS den Titel des Fensters, an dem das Programm identifiziert wird. Dieser Titel kann zum Beispiel auch über die Variable [active\_window] geholt werden, die den Titel des aktiven Vordergrundfensters enthält.

Anschliessend folgt ein in der bereits angesprochenen Makrosprache definierter Satz an Tastenkombinationen, die an das fernzusteuern Programm geschickt werden sollen.

Nehmen wir nun einmal diese Tastaturkommandos und Tastenkombinationen, welches wir bei SENDKEYS an den Notepad-Editor senden, genauer unter die Lupe:

Abschnitt:	A	B	C	D
Kommandos:	Hello World	^(alt)ds	test.txt(enter)	^(alt)(f4)

- Abschnitt A:
  - Hier schreiben wir einfach den Text "Hello World" in das Eingabefeld des Editors. Es muss darauf geachtet werden, dass der Cursor im richtigem Feld steht, wo der Text rein soll (das Eingabefeld hat beim Notepad nach dem starten IMMER den Eingabefokus!).
- Abschnitt B:
  - Jetzt wollen wir den Dialog zum Speichern der Datei öffnen. Dazu müssen wir ALT und D gleichzeitig drücken. Um eine Taste gedrückt zu halten, wird ihr ein Zirkumflex-Zeichen (^) vorangestellt. Da die ALT-Taste eine Sondertaste ist, wird diese in Klammern angegeben, damit der interne Interpreter für die SENDKEYS-Makrosprache weiß, das man eine Sondertaste ausgeben möchte.
  - Hier sei zu erwähnen, daß man, wenn man wirklich ein Klammerzeichen ausgeben möchte, diesem ein Tilde-Zeichen (~) voranstellen muß. Das Tilde-Zeichen erzwingt die Ausgabe des ihm nachfolgendem Zeichens, womit ^~ ein Zirkumflex an das Programm schickt und ~( eine offene Klammer. Um ein Tilde-Zeichen selbst an das fernzusteuern Programm zu senden, schreibt man ~~.
- Da wir nun die ALT-Taste gedrückt halten, senden wir direkt ein "d" hinterher. Mit ALT+D wählen wir also - via Tastenkombination - das Datei-Menü des Editors. Es sei hier auch zu erwähnen, das man, bevor man eine Anwendung mit RapidBATCH fernsteuert, sich vorher in der Anwendung das genaue Vorgehen selber anschaut, indem man seine Tastenkombinationen drückt. Beachten Sie bitte, daß direkt, nachdem das "d" an die Anwendung geschickt wurde, beide Tasten, also "d" und die ALT-Taste wieder automatisch "losgelassen" werden. Möchte man z.B. Mehrere Tasten gleichzeitig drücken, z.B. STRG, ALT und ENTF, so schreibt man als SENDKEYS-Makro "^(control)^(alt)^(del)".
- Um nun dem Speichern-Dialog zu laden, drücken wir noch direkt ein "s" hinterher.
- Wie Sie an dieser Stelle schon gemerkt haben, können "normale" Zeichen wie Buchstaben und Zahlen direkt angegeben werden, nur reine Sondertasten, für die es kein lesbares Zeichen gibt, werden geklammert geschrieben.
- Abschnitt C:
  - Hier "tippen" wir wieder einfach einen Text, in diesem Fall den Dateinamen "test.txt" ein, und bestätigen den Dialog mit der Taste Enter über den Sondertastenbefehl (enter).
- Abschnitt D:
  - Zum Schluss "drücken" wir noch einmal ALT+F4, um die Anwendung zu beenden. Auch F1 ist hier ein Sonderzeichen, und wird als (f1) kodiert. Es ist übrigens völlig Frei, ob Sie Groß- oder Kleinschreibung bei den Sondertastennamen angeben, (ALT) oder (Alt) wäre genauso richtig wie (alt).

Um eine Übersicht über die möglichen Sondertasten und ihrer Benennung zu erhalten, lohnt sich ein Blick in die Befehlsreferenz, in der auch nochmal die Syntax der SENDKEYS-Makrosprache Stichpunktartig erklärt wird.

Neben SENDKEYS bietet RapidBATCH noch eine weitere Möglichkeit, Programme fernzusteuern: Mit der MOUSEEVENT-Anweisung lassen sich alle nur erdenklichen Mausereignisse simulieren und damit fernsteuern, d.h. ein RapidBATCH-Script kann die volle Kontrolle über die Maus des Computers übernehmen und Klicks, Doppelklicks oder einfach nur Bewegungen simulieren.

Dieses Thema möchte ich jedoch nur kurz anreissen, da man aufgrund verschiedener Ausflösungen und Programme kein allgemeines Demo-Script, welches Mausereignisse vollzieht, durchführen kann. Scripts, die Mausereignisse simulieren, sollten Sie daher nur für den Eigengebrauch programmieren oder wenn Sie sich wirklich sicher sind, dass Ihre Mausroutine tadellos auch auf jedem System, wo Sie Ihr Script einsetzen, funktioniert.

Folgender Aufruf von MOUSEEVENT setzt den Mauszeiger exakt auf die Mitte des Bildschirms:

```
mouseevent 'move', ([ScreenWidth]/'2') # '|' # ([ScreenHeight]/'2')
```

Wie aus dem obigen Beispiel ersichtlich, erwartet MOUSEEVENT zwei Parameter. Zuerst einen Aktionsbezeichner, der MOUSEEVENT anweist, eine bestimmte Aktion durchzuführen, als zweites zwei Koordinatenwerte, die in der Form "X|Y" übergeben werden, wobei als Trennzeichen jedes beliebige, in der vordekliarten Systemvariablen [std\_sep] definierte Trennzeichen sein kann (wie auch bei Listen, die man an die LISTBOX-Funktion übergibt). Diese Koordinaten setzen den Mauszeiger pixelgenau auf die entsprechende Stelle auf dem Bildschirm, bzw. führen dort die entsprechend definierte Mausoperation durch.

Als Mausoperationen können folgende Werte an MOUSEEVENT übergeben werden:

- '0' oder 'MOVE' (Maus nur bewegen, kein Tastendruck)
- '1' oder 'CLICK\_LEFT' (einfachen Linksklick simulieren)
- '2' oder 'DBLCLICK\_LEFT' (doppelten Linksklick simulieren)
- '3' oder 'CLICK\_RIGHT' (einfachen Rechtsklick simulieren)
- '4' oder 'DBLCLICK\_RIGHT' (doppelten Rechtsklick simulieren)
- '5' oder 'DOWN\_LEFT' (Linke Maustaste gedrückt halten)
- '6' oder 'UP\_LEFT' (Linke Maustaste loslassen)
- '7' oder 'DOWN\_RIGHT' (Rechte Maustaste gedrückt halten)
- '8' oder 'UP\_RIGHT' (Rechte Maustaste loslassen)

## Bearbeiten der Registry

Mit der Version 5.0 ermöglicht RapidBATCH nun auch erstmals das direkte Bearbeiten der Windows-Registry. Die Registry ist eine mit Windows 95 eingeführte Datenbank, welche ein fester Bestandteil der Windows-Betriebssystemreihe geworden ist. Der hauptsächliche Zweck der Registry besteht darin, Konfigurationswerte und individualisierende Schalter für Programme abzulegen bzw. sogar benutzerorientiert zu speichern. Auch Informationen über Dateieindungstypen sowie alle nur erdenklichen Systemeinstellungen sind in der Registry hinterlegt. Damit entfällt auch die Vermüllung der Festplatte durch Konfigurations- und INI-Dateien fast gänzlich.

Editiert und betrachtet werden kann die Registry mit dem Windows-eigenen Werkzeug "regedit", welches bei jeder Windows-Version mitgeliefert wird.

Die Registry ist ähnlich dem Dateisystem von Windows aufgebaut, wobei es 5 so genannte Hauptschlüssel gibt. Ein Registrierungsschlüssel ist eine Art "Ordner" in der Registry, und kann theoretisch gesehen unendlich viele Unterschlüssel haben. In jedem Schlüssel können zusätzlich eine unbegrenzte Anzahl von Werten abgelegt werden, welche einen Namen und den entsprechend hinterlegten Wert haben.

Folgend eine Auflistung der 5 Hauptschlüssel und deren Bedeutung.

- HKEY\_CLASSES\_ROOT
- Hier werden Informationen über die verschiedenen Dateitypen gespeichert. Darunter fallen die Beschreibung, das Symbol und die verknüpfte Anwendung.
- HKEY\_CURRENT\_USER
- Hier werden Einstellungen und Informationen zu Software gespeichert, die nur für den aktuellen Windows-Benutzer gelten.

- HKEY\_LOCAL\_MACHINE
- Hier werden von allen Benutzern geteilte Einstellungen und Informationen gespeichert.
- HKEY\_USERS
- Enthält den Stamm aller Benutzerprofile auf dem Computer.
- HKEY\_CURRENT\_CONFIG
- Grundlegende Einstellungen des Betriebssystems.

Mit RapidBATCH lässt sich die Registry nun zunutze machen. Ob man nun via RapidBATCH bestehende Registry-Werte automatisiert verändert, um z.B. das Verhalten einer Software zu ändern oder zu bestimmen oder selbst einen eigenen Schlüssel für sein Script anlegt, um dort beispielsweise benutzerdefinierte Einstellungen und Schalter zu speichern, ist jedem selbst überlassen. Natürlich kommt es immer ganz auf den Zweck des jeweiligen Scripts an.

Zum Erstellen und Entfernen von Schlüsseln stellt RapidBATCH die Anweisungen NEWREGKEY und DELREGKEY zur Verfügung. NEWREGKEY erstellt einen Unterschlüssel, DELREGKEY entfernt einen Schlüssel samt Unterschlüssel und Werten aus der Registry. Als Parameter erwarten beide Anweisungen zuerst den Bezeichner des Hauptschlüssels, welcher durch die Werte 'CLASSES\_ROOT', 'CURRENT\_CONFIG', 'CURRENT\_USER', 'LOCAL\_MACHINE' und 'USERS' spezifiziert wird, sowie den Pfad des neu zu erstellenden Schlüssels. Folgendes Script erzeugt einen Schlüssel unter HKEY\_CURRENT\_USER mit dem Namen "Test".

```
rem Einen neuen Schlüssel erstellen
newregkey 'CURRENT_USER', 'Test'
if [errorcode] = '0'
    echo 'Erstellen von HKEY_CURRENT_USER\Test war erfolgreich.'
else
    echo 'Fehler beim Erstellen von HKEY_CURRENT_USER\Test.'
endif

rem Einen existierenden Schlüssel entfernen
delregkey 'CURRENT_USER', 'Test'
if [errorcode] = '0'
    echo 'Löschen von HKEY_CURRENT_USER\Test war erfolgreich.'
else
    echo 'Fehler beim Entfernen von HKEY_CURRENT_USER\Test.'
endif
```

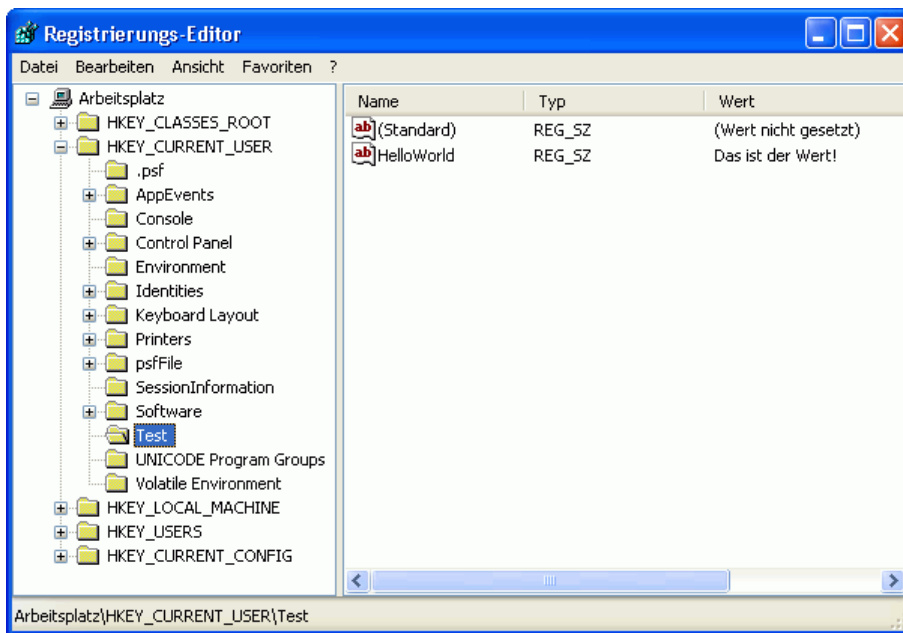
Wichtig beim Erstellen eines Schlüssels ist, dass die nächsthöhere Ebene des zu erstellenden Schlüssels bereits existiert, d.h. es besteht hier keine Äquivalenz zu MKDIR.

Um nun einen Wert in den Schlüssel selbst zu schreiben oder dem Schlüssel einen Wert hinzuzufügen bietet RapidBATCH die Anweisung LETREGKEY. LETREGKEY unterstützt dabei drei unterschiedliche Datentypen ("String", "Binary" und "Long" (DWORD)), die sich als Werte in der Registry ablegen lassen. Desweiteren erwartet die Anweisung den Hauptschlüsselnamen, den Schlüsselpfad, den Wertnamen sowie den zu setzenden Wert in der je nach Datentyp angegebenen Form. Folgendes Beispiel erzeugt einen neuen Wert "HelloWorld" im Schlüssel "Test" unter "HKEY\_CURRENT\_USER" (der Schlüssel wird hier vorher auch nochmal mit NEWREGKEY angelegt).

```
rem Testschlüssel anlegen
newregkey 'CURRENT_USER', 'Test'
if [errorcode] = '0'

    rem Wert in Registry schreiben
    letregval 'string', 'CURRENT_USER', 'Test', 'HelloWorld', 'Das ist der Wert!'

    rem War das Schreiben des Wertes erfolgreich?
    if [errorcode] = '0'
        echo 'Schreiben der Registry war erfolgreich!'
    else
        echo 'Fehler beim Schreiben der Registry'
    endif
endif
```



Erfolgreich geschriebener Wert in der Registry.

Es kann auch ein Wert direkt in den Schlüssel geschrieben werden; Hier lässt man einfach den Parameter für den Registry-Wertnamen leer.

```
letregval 'string', 'CURRENT_USER', 'Test', '', 'Das ist der Wert!'
```

Um einen Binärwert zu schreiben, wird bei der LETREGVAL genauso verfahren wie bei PUTFILE: Eine Liste an ASCII-Zeichencodes, bei der jeder Wert zwischen 0 und 255 liegen kann, also 8 Bit je Zeichen. Hier ein "Hello World" in binär:

```
rem Wert in Registry schreiben
[hello] = '72|101|108|108|111|32|87|111|114|108|100'
letregval 'binary', 'CURRENT_USER', 'Test', 'binary_HelloWorld', [hello]

rem War das Schreiben des Wertes erfolgreich?
if [errorcode] = '0'
    echo 'Schreiben der Registry war erfolgreich!'
else
    echo 'Fehler beim Schreiben der Registry'
endif
```



Ein deutlich erkennbares "Hello World" wurde binär in die Registry geschrieben!

Soll ein DWORD-Wert in die Registry geschrieben werden, können nur numerische 32-Bit Ganzzahlwerte in die Registry geschrieben werden.

Das Auslesen von Registry-Werten erfolgt auch hier wieder äquivalent über eine Funktion namens GETREGVAL. Parameter sind hier die selben wie bei LETREGVAL, nur dass hier natürlich kein Wert, dafür aber eine Rückgabeveriable angegeben werden muss.

Dieses Beispiel liest die oben geschriebenen Werte wieder aus der Registry aus:

```
rem Stringwert aus Registry lesen
getregval [text] = 'string', 'CURRENT_USER', 'Test', 'HelloWorld'
echo 'Stringwert in "HelloWorld": ' # [text]

rem Binärwert aus Registry lesen
getregval [data] = 'binary', 'CURRENT_USER', 'Test', 'binary_HelloWorld'
echo 'Gelesene ASCII-Liste aus "binary_HelloWorld": ' # [data]

rem ASCII-Codes in ASCII-Zeichen umwandeln und an Ausgabestring hängen
[i] = '0'
repeat
    [i] + '1'
    gettok [zeichen] = [data], [std_sep], [i]
    if [zeichen] ! ''
        getchr [zeichen] = [zeichen]
        [klartext] # [zeichen]
    endif
until [zeichen] = ''

rem Den Klartext ausgeben
echo 'Klartext aus "binary_HelloWorld": ' # [klartext]
```

Zu guter Letzt entfernen wir noch die geschriebenen Werte mit Hilfe der DELREGVAL-Anweisung aus der Registry.

```
rem "HelloWorld" entfernen
delregval 'CURRENT_USER', 'Test', 'HelloWorld'
if [errorcode] ! '0'
    echo 'Fehler beim Löschen von "HelloWorld"'
endif

rem "binary_HelloWorld" entfernen
delregval 'CURRENT_USER', 'Test', 'binary_HelloWorld'
if [errorcode] ! '0'
    echo 'Fehler beim Löschen von "binary_HelloWorld"'
endif
```

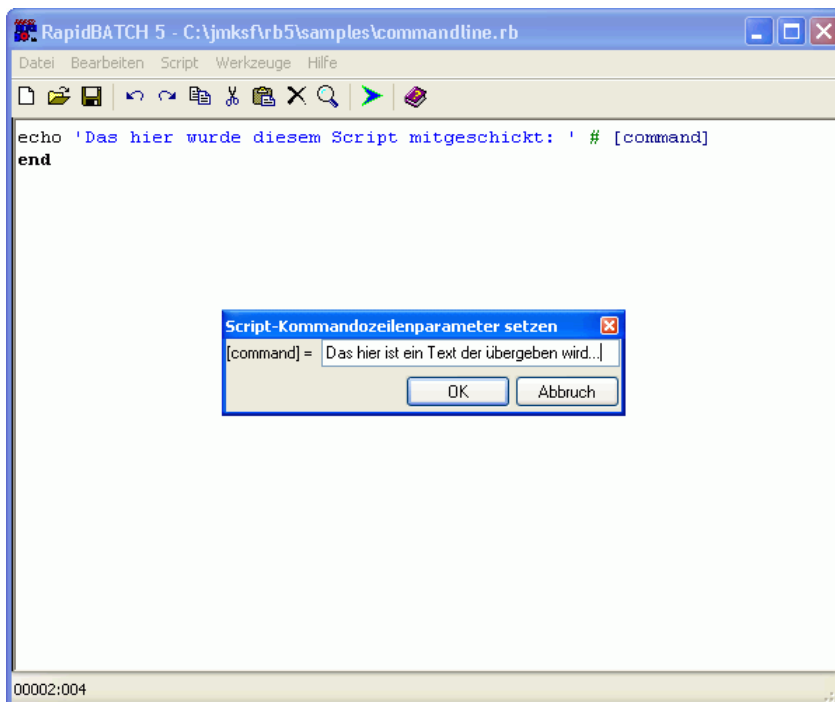
## Systemspezifische, vordeklarierte Variablen

Es mag vielleicht sein, dass der Abschnitt "Systemoperationen" nicht ganz passend ist, um alle vordeklarierten Variablen, die RapidBATCH bietet, zu behandeln, aber es haben auch sehr viele der vordeklarierten Variablen ihre Bestimmung bei der Programmierung von Scripts, die system-basierende Operationen ver- und bearbeiten. Bis jetzt haben Sie bereits viele der so genannten "vordeklarierten Variablen" kennen gelernt. Vordeklarierte Variablen werden vor der Ausführung eines jeden Scripts automatisch vom Interpreter deklariert. Bei einigen vordeklarierten Variablen ist der darin enthaltene Wert änderbar, bei anderen wird er bei jedem Aufruf der Variable direkt neu ermittelt.

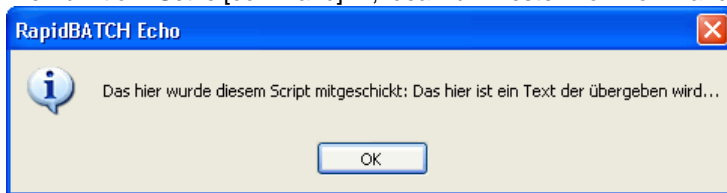
Ich werde in diesem Kapitel zwar nicht alle vordeklarierten Variablen behandeln, aber die wichtigsten, besonders solche, die für Systemoperationen von Bedeutung und Nutzen sind.

Oftmals ist es von großem Nutzen, ein Programm bzw. Optionen und Informationen an ein Programm über die Kommandozeile zu übergeben. Dies darf natürlich auch bei RapidBATCH nicht fehlen! Die Variable [command] enthält alle Kommandozeilenparameter, die hinter dem Dateinamen des Script-Files (wenn das Script mit dem Interpreter ausgeführt wird) bzw. bei compilierten Scripts alle an die EXE-Datei übergebenen Kommandozeilenparameter, übergeben wurden. Bestes Beispiel stellt folgendes Script dar. Es gibt einfach den Inhalt der Variablen [command] in einem Meldungsfenster aus. Im RapidBATCH Builder können Sie zum direkten Testen Ihrer Scripts die Funktion "Setze [command]..." im Menü "Script" benutzen.

```
echo 'Das hier wurde diesem Script mitgeschickt: ' # [command]
end
```



Die Funktion "Setze [command]...", ideal zum Testen von Kommandozeilen-Parametern...



...mit Erfolg!

Anmerkung: Die über [command] übergebenen Werte können direkt in ein Array transferiert werden, indem Sie die Prozedur `getcmdlineparms` aus der SYSTEM.RB Bibliothek verwenden.

Sie möchten einen beliebigen Wert für jedes andere Windows-Programm zugänglich machen? Mit RapidBATCH ist auch dies kein Problem. Durch Zuweisung eines beliebigen Wertes an die Variable [clipboard] wird dieser sofort in die Windows-Zwischenablage kopiert. Ein Auslesen von [clipboard] ermöglicht das sofortige Auslesen des Zwischenablagen-Inhaltes.

Ein einfaches Beispiel, welches sogar den Inhalt der Zwischenablage in einem anderen Programm einfügt:

```
rem Zeige mir den Inhalt der Zwischenablage!
echo 'Aktueller Inhalt der Zwischenablage: ' # [clipboard]

rem So und jetzt schreiben wir mal ein "Hello World" rein!
[clipboard] = 'Hello World by RapidBATCH'
echo 'Neuer Inhalt der Zwischenablage: ' # [clipboard]

rem Und jetzt via Fernsteuerung den Text mal im Notepad "einfügen"
shell [windir] # '\notepad.exe', 'show'
wait '500'
sendkeys [active_window]: 'Und das ist der Inhalt Ihrer Zwischenablage in einem anderem Programm: #271v#017'
end
```

Und da findet man auch direkt schon die nächste, vordeklierte Variable: [active\_window]! [active\_window] enthält den Titel des aktuellen Vordergrundfensters und kann idealerweise von SENDKEYS exzellent genutzt werden, wenn man den genauen Titel eines Fensters nicht kennt oder dieser variabel ist, wie es z.B. bei Word - oder anderen Office-Programmen - der Fall ist, wenn man ein Dokument geladen hat.

Auch die bereits kennen gelernte Variable [windir] wird im obigen Script benutzt; Sie enthält den absoluten Pfad des Windows-Systemverzeichnisses.



Auch das Auslesen von Systemdatum und Systemzeit ist in RapidBATCH kein Problem. Folgendes Script zeigt für 10 Sekunden die aktuelle Uhrzeit und das aktuelle Datum in einer INFOBOX sekundengenau an.

```
[i] = '0'
repeat
  [i] + '1'
  infobox 'Es ist ' # [time] # ' am ' # [day] # '.' # [month] # '.' # [year], 'show'
  wait '1000'
until [i] = '10'
end
```

Die Variable [time] enthält die Uhrzeit, wird also bei jedem Auslesen aktuell und neu gefüllt. Das Format der [time]-Variablen ist HH:MM:SS, also Stunden, Minuten und Sekunden. Anders ist es bei dem Datum. Da RapidBATCH ein international eingesetztes Softwareprodukt ist, wurde hier Wert auf einfache Datumsformatierung gelegt, und die drei Informationen "Tag", "Monat" und "Jahr" in drei Variablen, [day], [month] und [year] aufgeteilt. Man kann also durch einfaches "Zusammenbauen" des Datums mit dem Join-Operator (#) das Format völlig frei bestimmen.

Es ist 00:08:10 am 06.08.2005

Ausgabe der aktuellen Zeit und des aktuellen Tagesdatums im INFOBOX-Dialog.

Oftmals ist es auch nötig, ob nun bei der Ausgabe von Informationen oder direkt beim Schreiben von Dateien, beispielsweise Tabulatoren oder Zeilenumbrüche einzufügen. Da ein Tabulator oder Zeilenumbruch intern nichts anderes als ein weiteres ASCII-Zeichen ist, könnte man sich dieses auch mit der Funktion GETCHR holen, aber RapidBATCH hat auch bereits vordeklarierte Variablen, welche einem an dieser Stelle die Arbeit abnehmen: [new\_line], [tab], [crlf], [pipe] und [quot]. [new\_line] und [crlf] stellen jeweils Zeilenumbrüche dar, allerdings mit einem wichtigem Unterschied. Unter Windows-Systemen ist es besonders bei Texten üblich, Zeilenumbrüche mit zwei anstatt mit einem Zeichen einzuleiten. Diese heißen "carriage return" und "line feed" (ASCII-Zeichencodes 13 und 10, daher auch [crlf], Carriage Return + Line Feed). Diese beiden Begriffe kommen noch aus der Urzeit der Computer, wo die Ausgabe aller Programme noch vornehmlich auf Nadeldruckern erfolgte, daher "carriage return", was übersetzt soviel heisst wie "Schreibkopfrücklauf" und "line feed", was nicht mehr als "Zeilenvorschub" heisst. I.d.R. reicht meist aber ein einfaches [new\_line], welches nur ein "line feed" bewirkt, um einen Zeilenumbruch darzustellen. [tab] enthält ein Tabulatorzeichen, [pipe] das Zeichen "|", welches auch als RapidBATCH Standard Separator bei Listen verwendet wird. Die Variable [quot] enthält ein einfaches Anführungszeichen, welches in RapidBATCH selber zur Ein- und Ausleitung von Strings im Script-Code verwendet wird, und daher nicht direkt ausgegeben werden kann. Um dieses Zeichen selber in einem String zu verketteten, verwenden Sie [quot], welche dieses Zeichen (') enthält.

```
rem Zeilenumbrüche
echo 'Dies ist' # [new_line] # 'ein Zeilenumbruch in ' # [crlf] # 'zwei Versionen'

rem Tabulatoren
echo 'Ein' # [tab] # 'Tabulator!'

rem Pipe-Zeichen
echo 'Das als ' # [std_sep] # ' definierte Zeichen ist Standardmäßig ein ' # [pipe] # '
;)'

rem Einfaches Anführungszeichen
echo 'echo ' # [quot] # 'Dies ist der Code für eine einfache Ausgabe' # [quot] # [crlf] #
'end'
```

Bereits in vorherigen Abschnitten haben Sie die Variable [errorcode] kennen gelernt. Diese werden Sie auch am häufigsten zur Abfrage von Fehlern verwenden. [errorcode] gibt in fast allen Fällen (bis auf Ausnahmen bei SHELL und CALL) die Werte '0' für "kein Fehler" bzw. "wahr" und '-1' für "Fehler" bzw. "falsch" zurück. Um eine IF- oder UNTIL-Bedingung etwas professioneller zu realisieren, wurden während der Entwicklung von RapidBATCH 5 noch die Variablen [true] und [false] hinzugenommen. [true] enthält standardmäßig den Wert '0', [false] standardmäßig den Wert '-1'. Eine IF-Abfrage kann also zum besseren Verständnis sowie zur erhöhten Dynamik so programmiert werden:

```
chdir 'C:\test'

if [errorcode] = [false]
  echo 'Fehler, das Verzeichnis wurde nicht gefunden!'
else
  echo 'Das Verzeichnis wurde erfolgreich gewechselt.'
endif
```

Zu guter Letzt noch eine Variable, die im wahrsten Sinne des Wortes gar nichts tut: [null]!

Diese Variable hat u.a. den Zweck, Funktions-Rückgabewerte abzufangen, die nicht benötigt werden, denn nicht immer wird der Rückgabewert einer Funktion wirklich gebraucht. Bei Zuweisung an die Variable [null] wird der Wert im Speicher direkt verworfen, muss also nicht erst in eine Pseudo-Returnvariable kopiert werden, was wiederum unnützen Speicher-



verbrauch zur Folge hat. Des weiteren kann [null] auch als Parameter bei Anweisungen oder Funktionen übergeben werden, wenn man einen Leerstring übergeben will. [null] ist also die aussagekräftigere Alternative zur Angabe eines Leerstrings.

## Benutzerdefinierte Dialogprogrammierung

### Erstellung von Dialogfenstern

Mit RapidBATCH 5 ist es auch erstmals möglich, eigene Dialogfenster direkt in RapidBATCH zu implementieren. RapidBATCH verwendet dazu ein sehr einfaches, aber ausgeklügeltes System, welches es möglich macht, Dialogfenster zu erstellen und Dialogfensterelemente zu verändern und auszulesen.

Mit nur insgesamt fünf verschiedenen Anweisungen bzw. Funktionen haben Sie somit die Entwicklung eigener Windows-Programme mit und in RapidBATCH voll im Griff, und können so gut wie jede Art von Dialog und damit echte Windows-Anwendungen in RapidBATCH selber implementieren.

Im Grunde ist die Entwicklung von eigenen Dialogen in RapidBATCH eine simple und unkomplizierte Angelegenheit:

- Dialogelemente erzeugen
- Werte-/Eigenschaften der Dialogelemente setzen
- Dialogfenster anzeigen
- Dialog starten, auf Ereignisse reagieren und Werte-/Eigenschaften der Dialogelemente auswerten

Folgend dargestelltes Beispielscript führt genau diese Schritte durch; Es erstellt einen Dialog mit einem einzeiligen Eingabefenster und einem Button. Klickt der Benutzer den Button, so wird der im Textfeld eingetragene Text in einem Meldungsfenster ausgegeben.

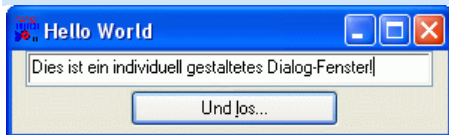
```
rem Widgets erzeugen
newdialog 'myDialog', 'DIALOG', '1|1|300|80'
newdialog 'myDialog:Eingabe', 'INPUT', '10|1|273|25'
newdialog 'myDialog:Los', 'BUTTON', '80|27|140|25'

rem Setzen der Werte/Eigenschaften
letdialog 'myDialog', 'CAPTION', 'Hello World'
letdialog 'myDialog:Eingabe', 'TEXT', 'Dies ist ein Text!'
letdialog 'myDialog:Los', 'CAPTION', 'Und &los...'

rem Dialog anzeigen
letdialog 'myDialog', 'VISIBLE', [true]

repeat
    rem Dialog starten
    rundialog [event] = '0'

    rem Ereignis auswerten
    if [event] = 'click_myDialog:Los'
        rem Textfeld auslesen
        getdialog [text] = 'myDialog:Eingabe', 'TEXT'
        echo 'Der Wert im Textfeld lautet: ' # [text]
    endif
until [event] = 'close_myDialog'
end
```



Script mit individuell programmiertem Dialog

Ich werde Sie nun Schritt-für-Schritt in diese neue Welt der schier unbegrenzten Möglichkeiten einführen, denn dieses neue, sehr effektive Feature, eigene Dialoge und Fenster zu programmieren, macht RapidBATCH nicht nur zu einer starken Script- und Automatisierungssprache sondern auch zu einem schnellen und unkomplizierten Werkzeug zur Entwicklung von echten, individuellen Windows-Anwendungen aller Art.

Anmerkung: Folgend werde ich des öfteren das Wort "Widget" verwenden. Als Widget (eine Abkürzung für "Window Gadget", zu deutsch "Fenster Ding") bezeichne ich hier sowohl Dialogfenster als auch Dialogelemente wie z.B. Buttons, Eingabefelder usw.. Die Unterschiede zwischen Dialogfenstern und Dialogelementen werden später genauer erläutert.

Hinweis: Mit der RapidBATCH Professional Edition haben Sie die Möglichkeit, Dialogfenster schnell & unkompliziert mit dem mitgelieferten Werkzeug "RapidBATCH Visual Dialog Designer" zu erstellen. Widgets können hier in einer Echtzeitumgebung erstellt, angeordnet und bearbeitet werden. Ist der Dialog fertig entworfen, generiert der Assistent den dafür nötigen RapidBATCH Code, den Sie lediglich um die Ereignisbehandlung der einzelnen Widgets erweitern brauchen. Zum Verständnis der Programmierung eigener Dialogfenster sollten Sie aber trotzdem dieses Kapitel durcharbeiten.

Der "Visual Dialog Designer" selbst ist komplett zu 100% in RapidBATCH geschrieben worden!

Um Widgets zu erzeugen verwenden wir die Anweisung NEWDIALOG. NEWDIALOG benötigt, wie im Beispielscript zu sehen, als Parameter ein so genanntes Widget-Label, welches man auch als Namen für das Widget ansehen kann; Über diesen Namen kann das Widget später angesprochen werden.

Des weiteren erwartet NEWDIALOG einen Widget-Typ, der angibt, welche Art von Widget wir erzeugen möchten (z.B. Dialogfenster, Button, einzeliges Eingabefeld, mehrzeiliges Eingabefeld, usw.) sowie die Pixelkoordinaten und -dimensionen, um das Widget zu platzieren. Die Koordinaten werden dabei durch den bereits bekannten Standard-Listenseparator (normalerweise ein "|" -Zeichen (Pipe)) als String in der Form "X-Koordinate|Y-Koordinate|Breite|Höhe" angegeben.

In unserem Beispielscript erzeugen wir also mit der Anweisung

```
newdialog 'myDialog', 'DIALOG', '1|1|300|80'
```

ein neues Widget vom Typ "DIALOG" (sprich: ein Dialog-Basisfenster) mit einer Breite von 300 Pixel und einer Höhe von 80 Pixel am linken oberen Bildschirmrand (X- und Y-Koordinaten 1|1). Dieses Dialogelement wird unter dem Namen "myDialog" angelegt, über den wir später auf das Element zugreifen können.

Die zwei darauffolgenden NEWDIALOG-Anweisungen

```
newdialog 'myDialog:Eingabe', 'INPUT', '10|1|273|25'
```

```
newdialog 'myDialog:Los', 'BUTTON', '80|27|140|25'
```

erstellen ein einzeliges Eingabefeld (Typ: "INPUT") und einen Button (Typ: "BUTTON") auf unserem Dialog-Basisfenster "myDialog".

Dass diese Widgets auf dem Basisdialog "myDialog" platziert werden, ist an der Voranstellung von "myDialog:" beim Widget-Label erkennbar. Die Widgets werden daher unter genau diesem Dialoglabel angelegt; Sinn und Zweck des ganzen ist, dass jedes Dialogelement auf einem Dialog einmalig (d.h. mit einem einmaligen Namen) sein muss, denn es können auch zwei oder mehr Dialogfenster zur selben Zeit angezeigt werden. Diese Dialogelemente werden als "Child-Elemente" bezeichnet, sie stellen sozusagen die Kinder des Dialogs "myDialog" dar, da sie auf "myDialog" angezeigt und diesem Basisfenster hierarchisch untergeordnet sind.

Die Angabe der Pixelkoordinaten ist bei allen Child-Widgets relativ zur linken oberen Ecke des Fensters.

Nachdem wir nun die Widgets erzeugt haben, setzen wir für diese verschiedene Werte, in unserem Fall bekommt das Dialogfenster den Titel "Hello World", der Button die Beschriftung "Und los..." und das Textfeld den Text "Dies ist ein Text!".

```
rem Setzen der Werte/Eigenschaften
letdialog 'myDialog', 'CAPTION', 'Hello World'
letdialog 'myDialog:Eingabe', 'TEXT', 'Dies ist ein Text!'
letdialog 'myDialog:Los', 'CAPTION', 'Und &los...'
```

Alle Werte werden mit Hilfe der Anweisung LETDIALOG gesetzt. LETDIALOG erwartet als Parameter das Label des Widgets, bei dem eine Eigenschaft gesetzt werden soll, die Bezeichnung der Eigenschaft, die gesetzt werden soll, sowie den entsprechenden Wert. Der Eigenschaftsbezeichner "CAPTION" setzt die Beschriftung eines Widgets, die Eigenschaft "TEXT" den Text im Editierbereich von Eingabefeldern; Welche Eigenschaften unterstützt werden ist von Widget zu Widget unterschiedlich; Ein Listenfeld kann beispielsweise keine Beschriftung (CAPTION) haben, ein Dialogfenster kann keine Liste eines Listenfeldes aufnehmen. Es ist also vom Typ des Widgets abhängig, was wie gesetzt wird. Wir werden jedoch später genauer darauf eingehen.

Mit der Zeile

```
letdialog 'myDialog', 'VISIBLE', [true]
```

machen wir zu guter letzt noch das Fenster sichtbar; Nur Dialogfenster-Widgets sind nach ihrer Erstellung mit NEWDIALOG unsichtbar, da es nicht empfehlenswert ist, das Basisfenster schon zur Anzeige zu bringen wenn noch nicht alle Dialogelemente erzeugt und gesetzt wurden.

Anmerkung: Wenn Sie eigene Dialoge in RapidBATCH programmieren, stellen Sie bitte immer sicher, dass das Basisfenster auf sichtbar gesetzt wird, bevor Sie die Funktion RUNDIALOG (zu der wir gleich kommen werden) aufrufen; Ist dies nicht der Fall, wartet das Script solange, bis ein Ereignis auf dem nicht-sichtbaren Fenster geschieht; Und das wird nie der Fall sein, denn wie soll man zum Beispiel auf einem nicht-sichtbaren Fenster etwas anklicken?

Die soeben schon erörterte Funktion RUNDIALOG startet nun eine interne Warteschleife, die auf Ereignisse vom Dialogfenster wartet.

```
rundialog [event] = '0'
```

Wenn nun beispielsweise ein Button gedrückt wurde, wird ein Ereignis (Event) ausgelöst. Die Funktion gibt in diesem Fall einen String im Format "Ereignis\_Widget-Label" zurück. In unserem Script wird z.B. der String "click\_myDialog:Los" zurückgegeben, wenn der "Und Los..."-Button, den wir zuvor definiert haben, angeklickt wurde.

Die bei RUNDIALOG als Parameter übergebene Zahl ist die Anzahl an Millisekunden, die gewartet wird, bis ein optionaler Timeout eintritt (1 Sekunde entspricht 1000 Millisekunden). Dies bedeutet, dass das Fenster nur für eine bestimmte Zeit auf Ereignisse wartet. Tritt kein Ereignis ein, wird der Timeout ausgelöst - die Funktion gibt dann einen Leerstring zurück. In unserem Fall aber ist der Millisekunden Wert 0, d.h. RUNDIALOG hat keinen Timeout und wartet solange, bis irgendetwas mit oder auf dem Dialog passiert.

Im darauffolgenden IF-Block überprüfen wir nun diesen von RUNDIALOG zurückgegebenen Event-String; Wenn also "click\_myDialog:Los" eintrat, lesen wir mit der GETDIALOG-Funktion den Text des Textfeldes aus und geben ihn mit Hilfe der ECHO-Anweisung aus.

```
getdialog [text] = 'myDialog:Eingabe', 'TEXT'
echo 'Der Wert im Textfeld lautet: ' # [text]
```

GETDIALOG ist sozusagen die umgekehrte Version von LETDIALOG: Hier lesen wir eine bestimmte Eigenschaft des ebenfalls bei der Funktion spezifizierten Dialoglabels aus, in diesem Fall die Eigenschaft "TEXT".

Mit der abschließenden UNTIL-Anweisung wird gewährleistet, dass RUNDIALOG solange wieder neu aufgerufen wird, bis der Benutzer den Schließen-Button des Basisfensters klickt. Ist dies der Fall, gibt RUNDIALOG den Ereignis-String "close\_myDialog" zurück, auf den wir entsprechend abprüfen:

```
repeat
    rem Dialog starten
    rundialog [event] = '0'

    rem Ereignis auswerten
    if [event] = 'click_myDialog:Los'
        rem Textfeld auslesen
        getdialog [text] = 'myDialog:Eingabe', 'TEXT'
        echo 'Der Wert im Textfeld lautet: ' # [text]
    endif
until [event] = 'close_myDialog'
```

Ich denke, dass Sie hiermit nun das grobe Verarbeitungskonzept von benutzerdefinierten Dialogen in RapidBATCH verstanden haben. Im nächsten Abschnitt werden wir genauer auf die verschiedenen Dialogelemente sowie ihre Eigenschaften und Ereignisse eingehen.







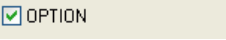

## Die Widgets im Überblick

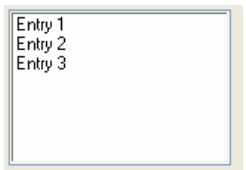
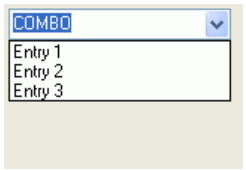
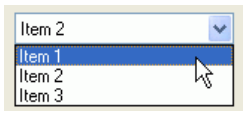

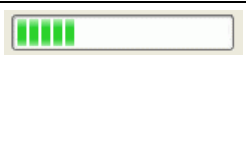
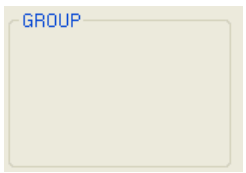
Zur Entwicklung von individuellen Dialogfenstern mit und in RapidBATCH stehen ihnen über 15 unterschiedliche Widgets mit zahlreichen Eigenschaften und Events zur Verfügung. In diesem Abschnitt wird jedes verfügbare Widget ausführlich vorgestellt.



### DIALOG

Ein Dialogfenster ist ein Widget, welches als Vaterwidget für alle weiteren Widgets (z.B. Buttons, Eingabefelder, usw.) agiert. Dieses Dialogfenster kann verschiedene Fensterstyles annehmen und besitzt eine Titelleiste mit einem frei definierbaren Text. Zusätzlich stellt dieses Widget auch Status- und Menübars bereit. Es können mehrere Dialogfenster zur selben Zeit angelegt und angezeigt werden.

	<p><b>BUTTON</b></p> <p>Ein Button (zu deutsch: Knopf) stellt eine Schaltfläche auf einem Dialog bereit. Buttons werden verwendet, um bestimmte Aktionen auszulösen. Sie können mit einem Text (CAPTION-Attribut) ausgestattet werden.</p>
	<p><b>IMAGEBUTTON</b></p> <p>Ein IMAGEBUTTON-Widget stellt eine zum Button-Widget äquivalente Schaltfläche auf einem Dialog bereit. Allerdings wird hier ein externes Bitmap-Bild (Dateiendung .BMP) anstatt eines Textes angezeigt.</p>
	<p><b>LABEL</b></p> <p>Labels (LABEL-Widgets) werden benutzt, um andere Widgets, wie z.B. Eingabefelder zu beschriften. Der jeweilige Text wird über das CAPTION-Attribut gelesen oder gesetzt.</p>
	<p><b>INPUT</b></p> <p>Ein INPUT-Widget stellt ein einzeliges Texteingabefeld bereit. Diese Widgets werden verwendet, um beispielsweise Zahlen oder einzelige Werte (z.B. Ortsnamen) einlesen zu können. Der Text kann über das TEXT-Attribut ausgelesen oder gesetzt werden, ebenso ist eine automatische Positionierung des Cursors bzw. des ausgewählten Textes mit dem SELPOS-Attribut möglich. Das SELECTION-Attribut gibt den selektierten Text zurück bzw. ersetzt ihn.</p>
	<p><b>PWD</b></p> <p>Ein PWD-Widget stellt ein einzeliges Passworteingabefeld bereit. Diese Widgets werden verwendet, um beispielsweise Passwörter oder Werte maskiert abzufragen, d.h. dass die Werte vom Anwender selbst bzw. auf dem Bildschirm nicht sichtbar sind. Die Benutzung dieses Widgets ist äquivalent zu der des INPUT-Widgets.</p>
	<p><b>EDIT, EDIT_LINEWRAP</b></p> <p>Ein EDIT-Widget stellt ein mehrzeiliges Texteingabefeld bereit. Diese Widgets können als kleine Texteditoren zur Bearbeitung und Darstellung von größeren Texten auf einem Dialog verwendet werden. Die Benutzung dieses Widgets ist äquivalent zu der des INPUT-Widgets.</p> <p>Widgets vom Typ EDIT_LINEWRAP haben die Besonderheit, dass nur eine vertikale Scrollbar existiert. Alle Zeilen, die über die Breite des EDIT_LINEWRAP-Widgets hinaussschießen werden dabei automatisch umgebrochen. Beim Auslesen des Widgets sind allerdings nur wirkliche Zeilenumbrüche, die der Benutzer manuell eingegeben hat, vorhanden. Vorteil von EDIT_LINEWRAP-Widgets ist, dass diese Widgets für die Eingabe von fließenden Texten, wie z.B. HTML-Code, äusserst effizient und ergonomischer sind. Der Text kann daher nur bei entsprechender Textlänge vertikal gescrollt werden.</p>
	<p><b>OPTION</b></p> <p>Ein OPTION-Widget stellt eine so genannte "Checkbox" bereit. Dies ist ein Kästchen, welches mit einem Häkchen versehen werden kann. OPTION-Widgets werden benutzt, um die Auswahl von Optionen zu ermöglichen, die der Benutzer ein- oder ausschalten kann. Mit Hilfe des CHECKED-Attributes kann der Häkchen-Status des OPTION-Widgets gesetzt bzw. ausgelesen werden (0 = gehakt, -1 nicht gehakt).</p>
	<p><b>RADIO</b></p> <p>Ein RADIO-Widget stellt einen so genannten Radio-Button bereit. Dies ist ein rundes Knöpfchen, welches mit einem Punkt "gesetzt" (aktiviert) werden kann. RADIO-Widgets werden meistens zu visuellen Gruppen zusammengefügt, von denen jeweils nur ein Widget gesetzt werden kann (daher auch der Name, da die Funktionsweise der Knöpfe der eines alten Kofferradios nachempfunden ist!). Das Setzen bzw. Auslesen des Status, ob das Widget "gehakt" ist, ist äquivalent zu der des OPTION-Widgets.</p>

	<p>LIST, LIST_SORTED, LIST_MULTI, LIST_MULTISORTED</p> <p>LIST-Widgets sowie deren Derivate, LIST_SORTED, LIST_MULTI und LIST_MULTISORTED, stellen ein Listenauswahlfeld, wie beim build-in LISTBOX-Dialog, als Widget bereit. LIST-Widgets stellen ihre Einträge so dar wie sie angegeben wurden, LIST_SORTED-Widgets sortieren die Einträge alphabetisch. Zum Füllen bzw. Auslesen der LIST-Widgets wird das ITEMS-Attribut verwendet, welches eine Liste von Einträgen, die jeweils durch Pipe-Zeichen von einander getrennten sind, verwendet. Der ausgewählte Eintrag wird mit dem SELECTION-Attribut gelesen bzw. bestimmt.</p> <p>LIST_MULTI- und LIST_MULTISORTED-Widgets unterscheiden sich von den LIST- und LIST_SORTED-Widgets insofern, dass anstatt eines einzelnen Listeneintrags gleich mehrere oder alle Einträge selektiert werden können. Das SELECTION- bzw. SELINDEX-Attribut gibt in diesem Fall eine Liste an selektierten Einträgen zurück.</p>
	<p>COMBO</p> <p>COMBO-Widgets stellen eine Mischung aus INPUT-Widget und LIST-Widget dar. Es kann ein Text in das Eingabefeld editiert werden oder aus der Dropdown-Box, die durch Drücken des kleinen Buttons rechts neben dem Eingabefeld aufklappt, ausgewählt werden. Die Übergabe der Eintragsliste erfolgt äquivalent zum LIST- und LIST_SORTED-Widget. Zudem können alle Operationen wie Text lesen/setzen, Cursor setzen oder markierten Text lesen/ersetzen genau wie beim INPUT-Widget angewandt werden.</p>
	<p>STATIC_COMBO</p> <p>STATIC_COMBO-Widgets sehen genau wie COMBO-Widgets aus, allerdings lässt sich der gewählte Wert nicht editieren oder verändert. Es kann also nur aus einer statischen und vordefinierten Menge von Werten gewählt werden.</p>
	<p>IMAGE</p> <p>Ein IMAGE-Widget ermöglicht das Anzeigen eines Bildes auf einem Dialogfenster. Das Bild wird mit Hilfe des IMAGE-Attributes gesetzt und automatisch auf die jeweilige Größe skaliert. Es können derzeit nur extern gespeicherte Bitmap-Bilder (Dateierweiterung .BMP) verwendet werden.</p>
	<p>PROGRESS</p> <p>PROGRESS-Widgets (Progressbars) werden verwendet, den Fortschritt einer Operation visuell als einen prozentualen Anteil darzustellen. Der prozentuale Anteil des Fortschrittsbalkens kann auf einen beliebigen Wert gesetzt werden.</p>
	<p>GROUP</p> <p>GROUP-Widgets (Groupboxes) werden verwendet, um andere Widgets visuell zu gruppieren, d.h. eine gewisse Zusammengehörigkeit einer Widget-Gruppe darzustellen. Diese Widgets haben nur eine rein visuelle Funktion. Der Titel der Groupbox wird mit Hilfe des CAPTION-Attributes gesetzt.</p>

## Entwicklung einer Beispielanwendung

Um einmal die Implementierung einer kleinen, kompletten Anwendung, welche mit einem benutzerdefinierten Dialogfenster arbeitet, zu demonstrieren, werden wir in diesem Kapitel einen Zahlensysteme-Rechner programmieren. Dieser "Bin/Okt/Hex Umrechner", wie ich ihn nenne, soll Dezimalzahlen in Binärzahlen, Oktalzahlen und Hexadezimalzahlen und umgekehrt umrechnen können.

Zur Berechnung der Zahlensysteme verwenden wir die Funktionen "base\_to\_decimal" und "decimal\_to\_base" aus der Bibliothek MATH.RB, da der Hauptschwerpunkt dieses Beispiels nicht beim Umrechnen der Zahlen, sondern beim Umgang mit den Dialogwidgets liegt.



Vorab aber trotzdem ein wenig Theorie am Beispiel der Umrechnung vom Dezimalzahlensystem in das Binärzahlensystem. Das Dezimalzahlensystem verwenden wir in unserem täglichen Gebrauch von Zahlen. Dezimalzahlen bestehen daher aus Ziffernfolgen von 0-9, sind also zur Basis 10 gehalten. Das Binärzahlensystem kennt nur zwei Ziffern, 0 und 1, und ist zur Basis 2 gehalten (daher auch manchmal der Begriff "Dualzahlensystem"). Auf niedrigster Ebene besteht jedes Programm, jedes Zeichen und jeder Wert, der mit oder von einem Computer verarbeitet wird, aus Millionen solcher Folgen wie z.B. "10011010010", was in dezimaler Schreibweise dem Wert "1234" entspricht. Errechnen lässt sich dieser binäre Wert recht simpel, indem man den Ausgangswert 1234 jeweils durch die Basis des Zahlensystems teilt, bei Dualzahlen jeweils durch 2. Erhält man einen Restwert mit Nachkommastelle, so wird eine 1 als Ziffer gesetzt, andernfalls eine 0. Kommt letztendlich als Restwert 0 heraus, ist die Zahl vollständig ins andere Zahlensystem übertragen (nachdem man sie noch einmal umgedreht hat (da die berechnete Zahl von rechts nach links aufgebaut wird)).

Unser Konvertierungsprogramm soll nun wie folgt aufgebaut werden: Ein Dialogfenster mit zwei Eingabefeldern, wobei das obere für die Aus- und Eingabe von Dezimalzahlen und das untere für die Aus- und Eingabe von Zahlen im jeweiligen Zahlensystem verwendet wird. In welches Zahlensystem umgerechnet wird, wird über ein Menü im Dialogfenster festgelegt, bei dem der jeweils aktuelle Umrechnungsmodus mit einem Häkchen markiert wird. Hier kann der Benutzer den Umrechnungsmodus wechseln. Das Programm soll jeweils vom Dezimal- ins eingestellte Zahlensystem als auch vom eingestellten Zahlensystem ins Dezimalsystem konvertieren können. Die Konvertierung soll "live" bereits während der Eingabe automatisch im jeweils umgekehrten Eingabefeld durchgeführt werden.

Diese ganzen Anforderungen an unser Programm stellen aber mit RapidBATCH gar kein Problem dar. In nicht mehr als 10 Minuten und einer Scriptlänge von etwa 120 Zeilen (inklusive entsprechender Kommentare und Leerzeilen) ist das komplette Programm fertig und lauffähig, wenn man bereits die entsprechende Erfahrung hat.

Im ersten Schritt bei der Entwicklung dieses Scripts entwerfen wir einen Prototypen. Ein Prototyp ist eine programmierte und lauffähige Version der Eingabemaske, jedoch ohne jegliche Funktionalität. Solch ein Prototyp ist auch eine bewährte Methode, die Widgets auf dem Fenster so anzuordnen, dass die Oberfläche auch ergonomisch und einheitlich aussieht.

```
rem *****
rem ** Prototyp "Bin/Okt/Hex Umrechner" **
rem *****

rem --- Erzeugen und Aufsetzen des Hauptfensters ---
newdialog 'umrechner', 'dialog', '1|1|270|100'

rem Fenstertitel und Style setzen
letdialog 'umrechner', 'caption', 'Bin/Okt/Hex Umrechner'
letdialog 'umrechner', 'style', 'single'

rem Menübaum erzeugen
[menü] # '&Modus:Dezimal -> &Binär|Dezimal -> &Okta|Dezimal -> &Hexadezimal;;'
letdialog 'umrechner', 'menu', [menü]

rem --- Erzeugung von Eingabefeldern und Labels ---

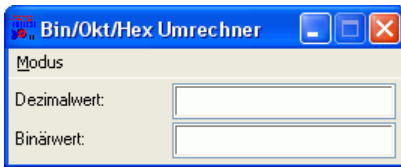
rem Label und Eingabefeld für Dezimalzahl
newdialog 'umrechner:l_dezimal', 'label', '5|8|100|22'
letdialog 'umrechner:l_dezimal', 'caption', 'Dezimalwert:'
newdialog 'umrechner:dezimal', 'input', '110|4|150|25'
letdialog 'umrechner:dezimal', 'font', 'Sans Serif|10|2'

rem Label und Eingabefeld für Zahl des jew. Zahlensystems
newdialog 'umrechner:l_konvert', 'label', '5|34|100|22'
letdialog 'umrechner:l_konvert', 'caption', 'Binärwert:'
newdialog 'umrechner:konvert', 'input', '110|30|150|25'
letdialog 'umrechner:konvert', 'font', 'Sans Serif|10|2'

rem Fenster sichtbar machen
letdialog 'umrechner', 'visible', [true]

rem Warten auf Ereignis (ohne Ereignisbehandlung)
rundialog [null] = '0'
end
```





Der bereits lauffähige Prototyp unseres Bin/Okt/Hex Umrechners

Wir werden nun Schritt-für-Schritt die einzelnen Abschnitte, in die der Prototyp eingeteilt ist, durchgehen.

Zuerst wird das Dialog-Basisfenster erzeugt mit einer Größe von 270x100 Pixel. In den darauf folgenden LETDIALOG-Anweisungen wird zuerst der Fenstertitel und anschließend ein so genannter Style, in unserem Fall der Style "SINGLE" für das Fenster gesetzt. Dieser Style ändert das Aussehen des Fensters, so dass kein Maximieren-Button mehr in der rechten oberen Ecke gegeben ist (da eine Maximierung des Fensters bei diesem kleinen Programm keinen Sinn ergibt). Andere Stylewerte können Sie der RapidBATCH-Sprachreferenz entnehmen.

Nachdem das Fensterstyle gesetzt wurde, weisen wir dem Fenster noch ein Menü zu. Dieses Menü wird als so genannter "Menu-Descriptor" übergeben. Dieser Descriptor definiert das Menü in einer einfachen Definitionssprache. In unserem Fall wird ein Hauptmenü "Modus" mit drei Untermenüpunkten ("Dezimal -> Binär", "Dezimal -> Oktal" und "Dezimal -> Hexadezimal") erzeugt. Mehr Informationen zum MENU-Attribut von LETDIALOG können Sie ebenfalls der Sprachreferenz entnehmen.

Im zweiten Abschnitt des Scripts erzeugen wir vier weitere Widgets, jeweils zwei Eingabefelder und zwei statische Texte, welche als Bezeichnungsfelder vor den Eingabefeldern angezeigt werden. Für die Eingabefelder setzen wir durch Verwendung des FONT-Attributes über LETDIALOG eine alternative Schriftart, hier "Sans Serif", 10 Punkt, Fett.

Auf Basis dieses Prototypen implementieren wir nun die Funktionalitäten zu unserem Programm.

Als erstes importieren wir die RapidBATCH Bibliotheken MATH.RB, welche die Konvertierungsfunktionen "decimal\_to\_base" und "base\_to\_decimal" bereitstellt, sowie die DIALOG.RB, welche ein paar nützliche Prozeduren beinhaltet, wie z.B. die Prozedur "centerdialog", die ein Dialogfenster auf dem Bildschirm zentriert. Das Importieren der Bibliotheken erfolgt als aller erstes im Programm mit Hilfe der INCLUDE-Anweisung.

```
rem --- Importieren von Bibliotheks-Hilfsfunktionen ---
include 'math.rb'
include 'dialog.rb'
```

Da die Funktionen decimal\_to\_base und base\_to\_decimal jeweils als zweiten Parameter einen String mit allen Ziffern des jeweiligen Zahlensystems erwarten (wobei die Anzahl der Ziffern gleichzeitig die Basis des Zahlensystems, in welches umgerechnet werden soll, definiert), deklarieren und füllen wir ebenfalls am Anfang unseres Prototypen-Programms entsprechende Variablen, die jeweils die Ziffern aller drei Zahlensysteme (Binär "01", Oktal "01234567" und Hexadezimal "0123456789ABCDEF") beinhalten. Eine Variable [aktuelle\_basis] verwenden wir später, um die jeweils aktuell umzurechnende Basis für das Umrechnungsprogramm zu setzen.

```
rem --- Globale Variablen (Basen und Ziffern der Zahlensysteme) ---
dec [bin], [okt], [hex], [aktuelle_basis]

rem Basis: 2 (Binärsystem)
[bin] = '01'
rem Basis: 8 (Oktalsystem)
[okt] = '01234567'
rem Basis: 16 (Hexadezimalsystem)
[hex] = '0123456789ABCDEF'
```

Bevor wir nun eine entsprechende Ereignisbehandlung für das Menü sowie die Eingabe und Umrechnung der Werte implementieren, programmieren wir zuerst zwei Prozeduren, um die Werte jeweils von Dezimal in das entsprechende Konvertierungssystem oder vom Konvertierungssystem in Dezimal umzurechnen. Der Einfachheit halber heißen diese Prozeduren berechne\_Zielwert und berechne\_Dezimalwert. Im Falle eines Fehlers (z.B. wenn eine ungültige Ziffer gefunden wurde) geben beide Umrechnungsfunktionen (base\_to\_decimal, decimal\_to\_base) den Wert '-1' zurück. Hier schreiben wir in das jeweilige Feld den Wert "Fehleingabe", um den Benutzer zu informieren, dass er einen ungültigen Wert umzurechnen versucht.

```
rem --- Implementation der Prozedur "berechne_Zielwert" ---
proc berechne_Zielwert
    dec [dezimal], [konvert]

    rem Zahlensystemwert aus Dezimalwert errechnen
    getdialog [dezimal] = 'umrechner:dezimal', 'text'
```

```

    if [dezimal] != ''
        rem Wert berechnen (mit der decimal_to_base-Bibliotheksfunktion)
        decimal_to_base [konvert] = [dezimal], [aktuelle_basis]
        if [konvert] = '-1' [konvert] = 'Fehleingabe'
    endif
    letdialog 'umrechner:konvert', 'text', [konvert]
endproc

```

```

rem --- Implementation der Prozedur "berechne_Dezimalwert" ---

```

```

proc berechne_Dezimalwert
    dec [dezimal], [konvert]

    rem Dezimalwert aus Zahlensystemwert errechnen
    getdialog [konvert] = 'umrechner:konvert', 'text'
    if [konvert] != ''
        rem Wert berechnen (mit der base_to_decimal-Bibliotheksfunktion)
        base_to_decimal [dezimal] = [konvert], [aktuelle_basis]
        if [dezimal] = '-1' [dezimal] = 'Fehleingabe'
    endif
    letdialog 'umrechner:dezimal', 'text', [dezimal]
endproc

```

Bevor wir uns auch noch die Mühe machen und später in der Ereignisbehandlung drei mal dasselbe (für jeden Umrechnungsfall, den man im Menü setzen kann) programmieren, entwickeln wir noch eine weitere Prozedur `setze_Zahlensystem`, der wir als Parameter eine der drei Zahlensystemdefinitionen, die wir zuvor in einer der Variablen gespeichert haben, zuweisen. Diese Prozedur entfernt zugleich alle Häkchen aus dem Menü und setzt anschließend für den jeweils korrekten Menüpunkt das Häkchen erneut. Das Setzen eines Häkchens für einen Menüpunkt erfolgt über das bei der Menüzuweisung automatisch generierte Menülabel, welches im Format "Dialogname:Menu\_Menüpunkttext" generiert wird, und dem CHECKED-Attribut, welches auf [true] ('0') oder [false] ('-1') gesetzt werden kann.

Am Ende der Prozedur rufen wir direkt nochmal die oben implementierte Prozedur `berechne_Zielwert` auf, um direkt beim Auswählen eines Menüpunktes die Zahl, die im Dezimalfeld steht, in das ausgewählte Zahlensystem umzurechnen. Ausserdem setzen wir vorher noch die vorhin definierte Variable `[aktuelle_basis]`, die wiederum von `berechne_Zielwert` und `berechne_Dezimalwert` verwendet wird, auf das entsprechend übergebene Zahlensystem.

```

rem --- Implementation der Prozedur "setze_Zahlensystem" ---

```

```

proc setze_Zahlensystem: [zahlensystem]
    rem Häkchen aus Menüpunkten entfernen
    letdialog 'umrechner:Menu_Dezimal -> &Oktal', 'checked', [false]
    letdialog 'umrechner:Menu_Dezimal -> &Hexadezimal', 'checked', [false]
    letdialog 'umrechner:Menu_Dezimal -> &Binär', 'checked', [false]

    rem Zahlensystembezeichnung vor dem Eingabefeld sowie Häkchen im Menü setzen
    if [zahlensystem] = [bin]
        letdialog 'umrechner:l_konvert', 'caption', 'Binärwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Binär', 'checked', [true]
    elseif [zahlensystem] = [okt]
        letdialog 'umrechner:l_konvert', 'caption', 'Oktalwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Oktal', 'checked', [true]
    elseif [zahlensystem] = [hex]
        letdialog 'umrechner:l_konvert', 'caption', 'Hexadezimalwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Hexadezimal', 'checked', [true]
    endif

    rem Aktuelle Basis auf übergebenes Zahlensystem setzen
    [aktuelle_basis] = [zahlensystem]

    berechne_Zielwert
endproc

```

Jetzt ist die Anwendung schon fast fertig. Was jetzt noch fehlt, ist eine Ereignisbehandlungsroutine.

Dazu erweitern bzw. modifizieren wir den Bereich, in dem wir die Eingabefelder erzeugen, insoweit, dass wir das CHANGE-Ereignis, welches auftritt, wenn sich irgendetwas an dem Inhalt des Eingabefeldes verändert hat, einschalten. Bei der Erstellung eines Widgets vom Typ INPUT ist dieses Ereignis standardmäßig ausgeschaltet. Einschalten können wir es, indem wir bei LETDIALOG als Attributbezeichner den Wert EVENT\_CHANGE auf [true] ('0') setzen. Ein weiteres Setzen dieses Attributes auf [false] (oder '-1') bewirkt die erneute Ausschaltung des Ereignisses für das jeweilige Widget.

Nach der Erzeugung der Widgets sowie Einschaltung der CHANGE-Ereignisse rufen wir hier auch bereits einmal unsere Prozedur `setze_Zahlensystem` mit der Variablen `[bin]` als Parameter auf, um direkt als Default-Umrechnungssystem das Binärzahlensystem zu bestimmen.

```
rem --- Erzeugung von Eingabefeldern und Labels ---

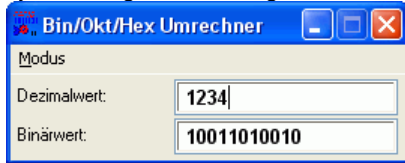
rem Label und Eingabefeld für Dezimalzahl
newdialog 'umrechner:l_dezimal', 'label', '5|8|100|22'
letdialog 'umrechner:l_dezimal', 'caption', 'Dezimalwert:'
newdialog 'umrechner:dezimal', 'input', '110|4|150|25'
letdialog 'umrechner:dezimal', 'font', 'Sans Serif|10|2'
letdialog 'umrechner:dezimal', 'event_change', [true]

rem Label und Eingabefeld für Zahl des jew. Zahlensystems
newdialog 'umrechner:l_konvert', 'label', '5|34|100|22'
newdialog 'umrechner:konvert', 'input', '110|30|150|25'
letdialog 'umrechner:konvert', 'font', 'Sans Serif|10|2'
letdialog 'umrechner:konvert', 'event_change', [true]
```

`setze_Zahlensystem [bin]`

Zu guter Letzt zentrieren wir das Fenster auf dem Bildschirm (mit Hilfe der Prozedur `centerdialog` aus der Bibliothek `DIALOG.RB`), setzen den Eingabefokus beim Programmstart auf das Dezimalzahlen-Eingabefeld und bringen den Dialog zur Anzeige.

In der Ereignisbehandlungsroutine, die wir, wie in unserem ersten Script mit selbst-definiertem Dialogfenster, als eine REPEAT...UNTIL-Schleife definieren, prüfen wir nun die 6 Ereignisse, die mit oder auf dem Dialog ausgelöst werden können ab und rufen die entsprechenden Prozeduren auf. Bei der Änderung eines Wertes in den Eingabefeldern wird jeweils die Prozedur `berechne_Zielwert` oder `berechne_Dezimalwert` (je nachdem in welchem Feld man sich befindet) eingegeben, bei Auswahl eines Menüpunktes wird die Prozedur `setze_Zahlensystem` mit dem entsprechenden Zahlensystem aufgerufen. Fertig ist unser selbstgeschriebener "Bin/Okt/Hex Umrechner"!



Der Bin/Okt/Hex Umrechner ist fertig!

Der gesamte Quellcode:

```
rem *****
rem ** Bin/Okt/Hex Umrechner **
rem ** Umrechnungsprogramm für Zahlensysteme **
rem *****
rem ** Copyright (C) 2005 by J.M. Meyer **
rem *****

rem --- Importieren von Bibliotheks-Hilfsfunktionen ---
include 'math.rb'
include 'dialog.rb'

rem --- Globale Variablen (Basen und Ziffern der Zahlensysteme) ---
dec [bin], [okt], [hex], [aktuelle_basis]

rem Basis: 2 (Binärsystem)
[bin] = '01'
rem Basis: 8 (Oktalsystem)
[okt] = '01234567'
rem Basis: 16 (Hexadezimalsystem)
[hex] = '0123456789ABCDEF'

rem --- Implementation der Prozedur "berechne_Zielwert" ---
proc berechne_Zielwert
    dec [dezimal], [konvert]

    rem Zahlensystemwert aus Dezimalwert errechnen
    getdialog [dezimal] = 'umrechner:dezimal', 'text'
```

```

    if [dezimal] != ''
        rem Wert berechnen (mit der decimal_to_base-Bibliotheksfunktion)
        decimal_to_base [konvert] = [dezimal], [aktuelle_basis]
        if [konvert] = '-1' [konvert] = 'Fehleingabe'
    endif
    letdialog 'umrechner:konvert', 'text', [konvert]
endproc

rem --- Implementation der Prozedur "berechne_Dezimalwert" ---
proc berechne_Dezimalwert
    dec [dezimal], [konvert]

    rem Dezimalwert aus Zahlensystemwert errechnen
    getdialog [konvert] = 'umrechner:konvert', 'text'
    if [konvert] != ''
        rem Wert berechnen (mit der base_to_decimal-Bibliotheksfunktion)
        base_to_decimal [dezimal] = [konvert], [aktuelle_basis]
        if [dezimal] = '-1' [dezimal] = 'Fehleingabe'
    endif
    letdialog 'umrechner:dezimal', 'text', [dezimal]
endproc

rem --- Implementation der Prozedur "setze_Zahlensystem" ---
proc setze_Zahlensystem: [zahlensystem]
    rem Häkchen aus Menüpunkten entfernen
    letdialog 'umrechner:Menu_Dezimal -> &Oktal', 'checked', [false]
    letdialog 'umrechner:Menu_Dezimal -> &Hexadezimal', 'checked', [false]
    letdialog 'umrechner:Menu_Dezimal -> &Binär', 'checked', [false]

    rem Zahlensystembezeichnung vor dem Eingabefeld sowie Häkchen im Menü setzen
    if [zahlensystem] = [bin]
        letdialog 'umrechner:l_konvert', 'caption', 'Binärwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Binär', 'checked', [true]
    elseif [zahlensystem] = [okt]
        letdialog 'umrechner:l_konvert', 'caption', 'Oktalwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Oktal', 'checked', [true]
    elseif [zahlensystem] = [hex]
        letdialog 'umrechner:l_konvert', 'caption', 'Hexadezimalwert:'
        letdialog 'umrechner:Menu_Dezimal -> &Hexadezimal', 'checked', [true]
    endif

    rem Aktuelle Basis auf übergebenes Zahlensystem setzen
    [aktuelle_basis] = [zahlensystem]

    berechne_Zielwert
endproc

rem --- Erzeugen und Aufsetzen des Hauptfensters ---
newdialog 'umrechner', 'dialog', '1|1|270|100'

rem Fenstertitel und Style setzen
letdialog 'umrechner', 'caption', 'Bin/Okt/Hex Umrechner'
letdialog 'umrechner', 'style', 'single'

rem Menübaum erzeugen
[menü] # '&Modus:Dezimal -> &Binär|Dezimal -> &Oktal|Dezimal -> &Hexadezimal;;'
letdialog 'umrechner', 'menu', [menü]

rem --- Erzeugung von Eingabefeldern und Labels ---

rem Label und Eingabefeld für Dezimalzahl
newdialog 'umrechner:l_dezimal', 'label', '5|8|100|22'
letdialog 'umrechner:l_dezimal', 'caption', 'Dezimalwert:'
newdialog 'umrechner:dezimal', 'input', '110|4|150|25'
letdialog 'umrechner:dezimal', 'font', 'Sans Serif|10|2'
letdialog 'umrechner:dezimal', 'event_change', [true]

```

```
rem Label und Eingabefeld für Zahl des jew. Zahlensystems
newdialog 'umrechner:l_konvert', 'label', '5|34|100|22'
newdialog 'umrechner:konvert', 'input', '110|30|150|25'
letdialog 'umrechner:konvert', 'font', 'Sans Serif|10|2'
letdialog 'umrechner:konvert', 'event_change', [true]

setze_Zahlensystem [bin]

rem --- Dialogfenster auf dem Bildschirm zentrieren & anzeigen ---
centerdialog 'umrechner'
letdialog 'umrechner', 'visible', [true]
letdialog 'umrechner:dezimal', 'focus', [true]

rem --- Ereignisbehandlungsroutine starten ---
repeat
  rundialog [ereignis] = '0'

  if [ereignis] = 'change_umrechner:dezimal'
    berechne_Zielwert
  elseif [ereignis] = 'change_umrechner:konvert'
    berechne_Dezimalwert
  elseif [ereignis] = 'click_umrechner:Menu_Dezimal -> &Binär'
    setze_Zahlensystem [bin]
  elseif [ereignis] = 'click_umrechner:Menu_Dezimal -> &Oktal'
    setze_Zahlensystem [okt]
  elseif [ereignis] = 'click_umrechner:Menu_Dezimal -> &Hexadezimal'
    setze_Zahlensystem [hex]
  endif

until [ereignis] = 'close_umrechner'
end
```

## Die DLL-Schnittstelle

### Verwendung externer Funktionen

RapidBATCH verfügt jetzt über eine Schnittstelle zu DLLs und ist damit leicht erweiterbar. Um diese Schnittstelle einfach und effizient zu halten, werden spezielle Anforderungen an die aufzurufenden DLLs gestellt. Es kann daher nicht jede DLL benutzt werden, sondern die DLLs müssen speziell zur Benutzung mit RapidBATCH programmiert sein.

Es gibt inzwischen einige fertige DLLs für RapidBATCH, die kostenlos benutzt werden können. Einige von ihnen finden sich bereits im aktuellen RapidBATCH Softwarepaket, andere lassen sich über das Internet meist kostenlos beziehen. Die Funktionen und wie sie benutzt werden ist je nach DLL unterschiedlich.

An dieser Stelle daher nur einige allgemeine Hinweise: Die EXT-Funktion versucht die angegebene DLL zu laden und dann die angegebene Funktion aufzurufen. Wenn nur der Name der DLL angegeben wird (ohne Pfad), dann wird die DLL zuerst im Anwendungsverzeichnis gesucht und dann im aktuellen Verzeichnis, im Windows- und im System32-Verzeichnis sowie in allen Verzeichnissen, die im Suchpfad aufgeführt sind. Die genaue Reihenfolge hängt vom Betriebssystem ab und kann bei Windows XP und Windows 2003 auch noch durch einen Registry-Eintrag geändert werden. Es wird aber immer zuerst das Anwendungsverzeichnis durchsucht. Dies ist das Verzeichnis, aus dem die EXE-Datei gestartet wurde (bei einem übersetzten Script) oder bei interpretierten Scripts das RapidBATCH-Verzeichnis (das Verzeichnis, welches die Datei RBI32.EXE, den RapidBATCH Interpreter, enthält).

Nach dem Aufruf sollte die interne Variable [errorcode] geprüft werden. Wenn die DLL oder die Funktion nicht gefunden wurde, dann wird [errorcode] auf '-1' ([false]) gesetzt. Ansonsten enthält [errorcode] einen von der DLL festgelegten Wert. Wichtig: Bei Funktionsnamen wird zwischen Groß- und Kleinschreibung unterschieden! Der Funktionsname muss daher genau so geschrieben werden wie in der Dokumentation der DLL vorgegeben. Beim Namen der DLL wird die Schreibweise dagegen nicht berücksichtigt.

### DLLs selber programmieren

Dieser Teil ist interessant für diejenigen, die selber DLLs programmieren möchten. DLLs können mit jeder Programmiersprache erstellt werden, die das Erstellen von DLLs mit C-Aufrufkonventionen erlaubt - auf jeden Fall also mit einem C-Compiler. Es folgt eine kurze Einführung in das Programmieren von DLLs und die RapidBATCH-Schnittstelle.

DLLs (Dynamic Link Librarys, zur Laufzeit einzubindende Bibliotheken) stellen Funktionen bereit, die von Anwendungen aufgerufen werden können. DLLs werden entweder sofort beim Start eines Programms geladen oder erst bei Bedarf. RapidBATCH verwendet die zweite Methode, d.h. die DLLs werden erst durch den EXT-Befehl geladen. Beim ersten Aufruf einer Funktion aus einer DLL wird die DLL in den Speicher geladen und initialisiert. Die DLL bleibt dann bis zum Programmende im Speicher und wird dann zusammen mit der Anwendung automatisch entladen.

Jede DLL muss eine Funktion "LibMain" enthalten. Diese Funktion wird automatisch beim Laden und Entladen der DLL aufgerufen. Die DLL kann in dieser Routine z.B. bei Laden Variablen initialisieren oder beim Beenden benutzte Objekte wieder freigeben oder temporäre Dateien löschen. Die Funktion LibMain muss beim Laden der DLL den Wert TRUE zurückgeben, sonst wird der Ladevorgang abgebrochen und die DLL wird sofort wieder entladen. RapidBATCH würde dann [errorcode] = '-1' liefern - so als wenn die DLL nicht gefunden wurde. Beim Entladen wird der Rückgabewert der Funktion LibMain ignoriert.

Beim ersten Aufruf eines EXT-Befehls wird zuerst die DLL geladen (und damit LibMain ausgeführt), dann wird die Funktion gesucht und aufgerufen. Bei weiteren EXT-Befehlen, die dieselbe DLL benutzen, wird direkt die angegebene Funktion gesucht und aufgerufen.

Die Funktionen, die von RapidBATCH aufgerufen werden sollen, müssen wie folgt deklariert werden:

```
int __declspec(dllexport) _cdecl funktionsName( char** output, char** input, int input_count )
```

output ist dabei ein Zeiger auf die Rückgabeveriable, input ist ein Zeiger auf ein Array mit Eingabeparametern und input\_count enthält die Anzahl der übergebenen Eingabeparameter. Der für den Rückgabewert benötigte Speicher muss mit der Windows API-Funktion LocalAlloc() angefordert werden. Dieser Speicherbereich wird von RapidBATCH automatisch wieder freigegeben, um Speicherlecks zu verhindern. Im Fehlerfall kann \*output auch auf (char\*)NULL gesetzt werden. In diesem Fall liefert die EXT-Funktion einen leeren String zurück. Der Rückgabewert der Funktion wird in der

RapidBATCH-Variablen [errorcode] gespeichert. Im Erfolgsfall sollte daher 0 (Null) zurückgegeben werden oder im Fehlerfall eine positive Zahl als Fehlernummer.

Im Verzeichnis DLL\DemoDLL des RapidBATCH-Installationsverzeichnis findet sich eine Beispiel-DLL (DemoDLL.dll) mit Quelltext (DemoDLL.c) und einem Beispielscript (DemoDLL.rb), das diese DLL aufruft. Der Quelltext ist kommentiert und enthält noch einige weitere Informationen zum Thema RapidBATCH und DLLs.

Noch ein Hinweis zu den Funktionsnamen: Viele C-Compiler neigen dazu, die Funktionsnamen mit einem '\_'-Zeichen zu dekorieren. Meistens kann das über entsprechende Compileroptionen abgeschaltet werden. Wenn das nicht möglich ist, dann muss im RapidBATCH-Script der Name der Funktion mit einem vorangestellten '\_'-Zeichen verwendet werden.

## Ein Beispielprojekt

Hier ein kleines Beispielprojekt mit dem kompletten C-Quellcode einer einfachen DLL-Datei und dem entsprechenden RapidBATCH-Script, welches diese DLL verwendet. Beide Dateien finden sich auch im Verzeichnis DLL\DemoDLL des RapidBATCH Installationsverzeichnis.

An dieser Stelle ein großes Lob an Klaus Fischer (radix42) für das schreiben dieser Einführung und für die Entwicklung der zahlreichen Beispiel-DLLs.

```
// <windows.h> wird für alle Windows-Funktionen benötigt
#include <windows.h>

// <stdio.h> wird für die Funktion "sprintf" benötigt
#include <string.h>
#include <stdio.h>

//-----

// Dies ist die Start-/Stop-Routine einer jeden DLL.
// Diese Routine wird beim Laden und Entladen der DLL aufgerufen.
// Für die Funktion gilt die WINAPI Aufrufkonvention.

BOOL WINAPI __declspec(dllexport) LibMain( HINSTANCE hDLLInst, DWORD fdwReason, LPVOID
lpvReserved )
{
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
            /*
             * Dieser Teil wird ausgeführt, wenn die DLL geladen wird. Hier können
             * allgemeine Initialisierungsroutinen aufgerufen werden.
             * In diesem Beispiel wird eine MessageBox angezeigt, damit das Laden
             * der DLL verfolgt werden kann.
             */
            MessageBox( NULL, "DemoDLL.dll wurde geladen.", "RB5 DemoDLL",
MB_ICONINFORMATION );
            break;
        case DLL_PROCESS_DETACH:
            /*
             * Dieser Teil wird ausgeführt, wenn die DLL entladen wird. Hier wird
             * wieder aufgeräumt, wenn das nötig sein sollte.
             * In diesem Beispiel wird ebenfalls eine MessageBox angezeigt.
             */
            MessageBox( NULL, "DemoDLL.dll wird entladen.", "RB5 DemoDLL",
MB_ICONINFORMATION );
            break;
    }
    return TRUE;
}

//-----
```



```
/* Hier die erste Funktion: rbxGetTickCount
```

rbxGetTickCount ruft einfach die Windowsfunktion GetTickCount auf. Diese liefert die Anzahl Millisekunden seit dem Start von Windows. Damit kann sehr einfach die Dauer eines Vorgangs gemessen werden: Einfach vorher und nachher rbxGetTickCount aufrufen und die Differenz bilden.

ACHTUNG: GetTickCount liefert einen 32Bit Wert. Das reicht für ca. 49,7 Tage! Dann läuft der Zähler einfach über. Daher sollte in einer echten Anwendung überprüft werden, ob der zweite Werte kleiner als der erste ist, um darauf zu reagieren.

Alle Funktionen, die von RB aus aufgerufen werden sollen, müssen wie hier angegeben deklariert werden( bei anderen Compilern sieht die Syntax evtl. anders aus ). Wichtig sind aber immer "\_cdecl" und die Typen der Parameter.

Output enthält einen Zeiger auf eine Variable (die dann einen Zeiger auf eine Zeichenkette enthält). Input enthält einen Zeiger auf ein Feld von Zeigern auf Zeichenketten (die in RB übergebenen Parameter). InputCount enthält die Anzahl der übergebenen Parameter = die Anzahl der Elemente im Feld Input.

In dieser Funktion werden die übergebenen Parameter einfach ignoriert.

```
*/
int __declspec(dllexport) _cdecl rbxGetTickCount( char ** Output, char ** Input, int InputCount )
{
    // Zuerst prüfen, ob Output auf etwas zeigt.
    // Wenn nicht, dann hier abbrechen.
    // Der Wert -1 steht für einen RapidBATCH-Fehler - dies kommt normalerweise nicht
vor!
    // Es wird aber für alle Fälle geprüft, um auf der sicheren Seite zu sein.
    if( Output == NULL )
        return( -1 );

    // Einen Speicherbereich von Windows anfordern. "LPTR" bedeutet, dass der
    // Speicherbereich über die zurückgegebene Adresse ansprechbar ist und mit
    // Nullen gefüllt werden soll. Die zweite Angabe gibt die gewünschte Länge
    // in Bytes (Zeichen) an. Wichtig: Es muss immer mindestens ein Byte mehr
    // angefordert werden, als maximal Zeichen benutzt werden, da am Ende immer
    // noch ein Null-Byte angefügt wird. Da eine 32Bit-Zahl zurückgegeben wird,
    // sollten 15 Stellen reichen (15 Zeichen + 1 Null-Byte).
    // Die Adresse des Speicherbereichs wird in die Variable geschrieben, auf
    // die die Variable Output zeigt.
    *Output = LocalAlloc( LPTR, 16 );

    // Prüfen, ob die Speicheranforderung erfolgreich war. Wenn z.B. nicht genug
    // Speicher zur Verfügung stand, wird NULL zurückgegeben. In diesem Fall
    // ebenfalls hier abbrechen (mit Fehler -1. s.o.). Dieser Fall sollte
    // normalerweise ebenfalls nicht eintreten.
    if( *Output == NULL )
        return( -1 );

    // Jetzt kommt die klassische Ausgabe
    // Was alles wie mit "sprintf" geht, sollte bekannt sein (sonst K&R lesen)!
    // "GetTickCount" klappt lt. WinAPI immer, daher ist keine Prüfung auf
    // evtl. Fehler nötig und "GetTickCount" kann direkt in "sprintf" eingesetzt wer-
den.
    sprintf( *Output, "%d", GetTickCount() );

    // Das war's. Funktion beenden( "0" steht hier für "Erfolgreich", s.o. )
    return( 0 );
}
```

```
}

//-----

/* Hier die zweite Funktion: rbxCreateWindowList

rbxCreateWindowList erzeugt eine Liste der Überschriften aller sichtbaren Fenster.
Die Listenelemente werden jeweils durch die als Parameter übergebene Zeichenkette
getrennt. Zum Abrufen der Fensterliste wird die Windows-Funktion "EnumWindows"
verwendet, die für jedes Fenster eine sogenannte Callback-Funktion aufruft. Das ist
eine anwenderdefinierte Funktion, deren Adresse an "EnumWindows" übergeben wird.
Die Definition der Parameter und des Rückgabewerts ist festgelegt. Diese Funktion
wird von der Windows-Funktion "EnumWindows" für jedes Fenster aufgerufen. Dabei wird
jeweils eine Fensternummer (Window-Handle) übergeben. Diese Funktion wird im folgenden
definiert. Sie wird nicht exportiert, da sie nicht von RB aus benutzt wird und Windows
die Adresse ja beim Aufruf von "EnumWindows" übergeben bekommt.

Der Name der Funktion und die Namen der Parameter sind frei wählbar, es wird hier
der Einfachheit halber die Definition der Funktion in der Windows-API selbst benutzt.

Im Parameter "hwnd" wird jeweils eine Fensternummer übergeben. Der Parameter "lParam"
wird vom Anwenderprogramm an "EnumWindows" übergeben und von dort weiter an die
Funktion "EnumWindowsProc". Damit kann man der Callback-Funktion noch eigene Parameter
übergeben (alternativ könnten globale Variablen verwendet werden). In unserem Beispiel
wird in die Variable nicht benutzt.

Die Überschriften werden in einen vorher angeforderten Puffer geschrieben. Die Grösse
sollte ausreichen, trotzdem muss natürlich ein Überlauf abgefangen werden. Für die
Grösse wird eine Konstante per "#define" deklariert, so dass bei Bedarf nur eine Stelle
im Quelltext geändert werden muss.
*/

#define BUFFSIZE 10000 // 10000 Bytes sollten reichen, bei Bedarf erhöhen
#define TEXTSIZE 100 // 100 Bytes pro Titel, längere Titel werden abgeschnitten

// hier noch eine Fehlernummer
#define ERROR_BAD_INPUT 11

// Hier zwei globale Variablen, die auf den Puffer für die Überschriften
// und die Trennzeichenkette zeigen
char* gsSeparator = (char*)NULL;
char* gsBuffer = (char*)NULL;

//-----

/* Hier die Callback-Funktion.

In der Variablen hwnd wird jeweils eine Fensternummer übergeben. Der Parameter "lParam"
wird nicht benutzt (es wird NULL übergeben).
*/

BOOL CALLBACK EnumWindowsProc( HWND hwnd, LPARAM lParam )
{
    // Zuerst einige lokale Variablen
    char Title[TEXTSIZE];
    memset( Title, '\0', TEXTSIZE );

    // Titel des angegebenen Fensters holen
    // als max. Länge wird "TEXTSIZE-1" übergeben, um auf jeden Fall Platz für
    // das Null-Byte am Ende zu haben.
    GetWindowText( hwnd, Title, TEXTSIZE-1 );

    // Prüfen, ob der Titel nicht leer ist, sonst die Funktion beenden
    if( *Title != 0 )
    {

```

```
// Prüfen, ob das Fenster sichtbar ist, sonst dieses Fenster
// ignorieren und das evtl. nächste anfordern.
if( IsWindowVisible( hwnd ) )
{
    // Das Fenster hat eine nicht leere Überschrift und ist sichtbar,
    // also in die Liste aufnehmen.

    // Zuerst prüfen, ob die Liste schon etwas enthält
    if( *gsBuffer != 0 )
        // Ja, die Liste ist nicht leer,
        // also zuerst die Trennzeichenkette anhängen
        strcat( gsBuffer, gsSeparator );
    // Dann den Fenstertitel des übergebenen Fensters anhängen
    strcat( gsBuffer, Title );

    // Jetzt wird geprüft, ob noch genügend Platz im Puffer für
    // eine weitere Trennzeichenkette und eine weitere Überschrift ist
    if( strlen( gsBuffer ) + strlen( gsSeparator ) + TEXTSIZE < BUFFSIZE
)
        // Ja, noch genug Platz, also TRUE zurückgeben,
        // um evtl. weitere Fensternummer anzufordern.
        return( TRUE );
    else
        // Nein, der Puffer ist fast voll. Die Aufzählung der Fenster
        // an dieser Stelle beenden( auch wenn noch ein kurzer
        // Fenstertitel Platz gehabt hätte ).
        return( FALSE );
    }
}

// TRUE bedeutet, das weitere Fensternummern geliefert werden sollen
//(soweit vorhanden)
return( TRUE );
}

//-----

/* Hier die eigentliche Funktion: rbxCreateWindowList

Diese Funktion benötigt einen Parameter: Den Zeiger auf eine Trennzeichenkette.
Sinnvollerweise wird hier die RB-Konstante [std_sep] oder [new_line] übergeben.

Diese Funktion ruft dann die Windows-Funktion "EnumWindows" auf, die dann für jedes
Fenster die oben definierte Callback-Funktion "EnumWindowsProc" aufruft.

*/

int __declspec(dllexport) _cdecl rbxCreateWindowList( char ** Output, char ** Input, int
InputCount )
{
    // Zuerst wieder die Prüfung der Parameter. Zusätzlich zu "Output" werden
    // jetzt aber auch "Input" und "InputCount" überprüft. Ausserdem darf die
    // übergebene Trennzeichenkette nicht zu lang sein (wäre sie z.B. "BUFFSIZE"
    // Zeichen lang, würde es beim zweiten Fenster, das in die Liste aufgenommen
    // werden soll, zu einem Pufferüberlauf kommen). Um die entsprechende Prüfung
    // in der Funktion "EnumWindowsProc zu sparen, wird einfach die Länge beschränkt.

    if( Output == NULL )
        return( -1 );

    // Einen Speicherbereich von Windows anfordern. Als Grösse wird hier
    // die Konstante "BUFFSIZE" benutzt. "LocalAlloc" füllt den Speicher-
    // bereich mit Nullen, das entspricht einer leeren Zeichkette.
    *Output = LocalAlloc( LPTR, BUFFSIZE );
}
```

```
// Prüfen, ob die Speicheranforderung erfolgreich war.
if( *Output == NULL )
    return( -1 );

// Jetzt wird die Anzahl der Parameter überprüft sowie die Trennzeichenkette
selbst
// "Input == NULL" und "*Input == NULL" dürften nicht eintreten, wenn "InputCount
== 1" gilt
// "***Input == 0" prüft auf eine leere Zeichenkette (ist das erste Zeichen == 0)
if( InputCount != 1 || Input == NULL || *Input == NULL || **Input == 0 || strlen(
*Input ) > TEXTSIZE )
{
    // Es wird eine leere Zeichenkette zurückgegeben (LocalAlloc hat ja den
    // angeforderten Speicherbereich mit Nullen gefüllt).
    // Der Rückgabewert der Funktion (hier z.B. 11) wird von RapidBATCH
    // in der Variablen [errorcode] gespeichert.
    return( ERROR_BAD_INPUT );
}

// Der Puffer ist verfügbar, die Trennzeichenkette ist korrekt
// Jetzt könnte die Funktion "EnumWindowsProc" aufgerufen werden.
// Allerdings hat die Funktion keinen Zugriff auf die Variablen
// und Parameter dieser Funktion, deshalb werden die benötigten
// Werte zusätzlich in globalen Variablen gespeichert
gsBuffer = *Output;
gsSeparator = *Input;

// Jetzt ist aber alles bereit, los geht's
// Der Rückgabewert der Funktion "EnumWindows" wird ignoriert,
// da er dem letzten Rückgabewert der Funktion "EnumWindowsProc"
// entspricht und das Ergebnis ("TRUE" oder "FALSE") in diesem Beispiel
// keine Rolle spielt.
// Der zweite Parameter (hier "NULL") wird von "EnumWindows" an unsere
// Callback-Funktion "EnumWindowsProc" weitergereicht. Da er nicht benötigt
// wird, wird hier "NULL" übergeben.
EnumWindows( EnumWindowsProc, (LPARAM)NULL );

// EnumWindows ruft jetzt unsere Funktion "EnumWindowsProc" mit allen
// Fensternummern auf und kehrt dann hierhin zurück.
// An dieser Stelle ist die Liste der Fensterüberschriften erstellt.
// Bleibt nur noch das Beenden dieser Funktion. Der Rückgabewert
// ist 0 (Null), um den erfolgreichen Abschluss anzuzeigen.

return( 0 );
}

//-----

// Das war's - viel Spaß beim Selbermachen...
```

Und das dazugehörige Aufruf-Script:

```
rem Dieses Skript demonstriert die Benutzung von DLLs mit dem EXT-Befehl
rem (es wird die DLL DemoDLL.dll verwendet).

rem Startzeit lesen. Es wird die Funktion rbxGetTickCount aus der
rem DLL DemoDLL.dll verwendet. Diese Funktion liefert die Zeit in
rem Millisekunden seit dem Start von Windows.

rem Bei diesem ersten Aufruf einer Funktion aus DemoDLL.dll wird
rem die DLL geladen - das wird über eine MessageBox signalisiert.
rem Bei weiteren Aufrufen wird die DLL im Speicher benutzt,
rem daher keine weiteren Meldungen der DLL.

ext [StartTime] = 'DemoDLL.dll', 'rbxGetTickCount'
```

```
rem Auf Fehler überprüfen
if [errorcode] ! '0'
    echo 'Fehler beim Aufruf von <rbxGetTickCount> in DemoDLL.dll'
    halt
endif

rem MessageBox anzeigen und Aktion des Benutzers abwarten.
rem Der Stil '262196' entspricht '4' + '48' + '262144',
rem d.h. ein '!' anzeigen (4) und die Auswahl Ja/Nein (48)
rem anbieten. Ausserdem soll die MessageBox immer im
rem Vordergrund bleiben (262144).
msgbox 'DemoDLL.rb', 'Entscheiden Sie sich!', '262196'

rem Stopzeit holen
ext [StopTime] = 'DemoDLL.dll', 'rbxGetTickCount'
rem Auf Fehler überprüfen
if [errorcode] ! '0'
    echo 'Fehler beim Aufruf von <rbxGetTickCount> in DemoDLL.dll'
    halt
endif

rem Zeitdifferenz berechnen, umrechnen in Sekunden.
[Differenz] = ([StopTime] - [StartTime]) / '1000'

rem Ergebnis anzeigen. Der Stil '64' zeigt das Symbol (i)
rem vor der Meldung an.
msgbox 'DemoDLL.rb', 'Sie haben ' # [Differenz] # ' Sekunden gebraucht, um sich zu ent-
scheiden.', '64'

rem Jetzt wird eine Liste der sichtbaren Fenster erstellt
rem mit der Funktion rbxCreateWindowList. Als Trennzeichen
rem wird ein Zeilenumbruch ([new_line]) verwendet, damit
rem das Ergebnis mit einer MessageBox angezeigt werden kann.
ext [WindowList] = 'DemoDLL.dll', 'rbxCreateWindowList', [new_line]
rem Auf Fehler überprüfen
if [errorcode] ! '0'
    echo 'Fehler beim Aufruf von <rbxCreateWindowList> in DemoDLL.dll'
    halt
endif

rem Liste mit einer MessageBox anzeigen
msgbox 'DemoDLL.rb', 'Es wurden folgende Fenster gefunden:' # [new_line] # [new_line] #
[WindowList], '64'

rem Jetzt wird noch die Fehlerprüfung getestet - dazu wird die
rem Funktion rbxCreateWindowList ohne Parameter aufgerufen. Das sollte
rem den Fehler 11 liefern (s. DemoDLL.c).
ext [WindowList] = 'DemoDLL.dll', 'rbxCreateWindowList'
rem Auf Fehler überprüfen
if [errorcode] ! '11'
    echo 'Unerwarteter Fehlercode beim Aufruf von <rbxCreateWindowList> in DemoDLL.dll'
    halt
endif

rem Korrekten Fehlercode melden
msgbox 'DemoDLL.rb', 'Die Parameterprüfung in <rbxCreateWindowList> hat geklappt.', '64'

rem Das war's.

rem DemoDLL.dll wird jetzt entladen, was ebenfalls mit einer MessageBox
rem angezeigt wird.

end
```





*Windows Scripting Technology*

# ***RapidBATCH 5 Sprachreferenz***



## Programmablauf und –steuerung

### DEC, RESET

#### Syntax:

```
DEC variable1 [,variable2 [ ... variablen ] ]  
RESET variable1 [,variable2 [ ... variablen ] ]
```

#### Beschreibung:

DEC deklariert eine oder mehrere neue Variablen im Script.

Wird DEC innerhalb einer Prozedur aufgerufen, so werden alle Variablen lokal deklariert. Bei einem Aufruf von DEC ausserhalb einer Prozedur sind die Variablen global im gesamten Script verfügbar. DEC kann auch zum Vordimensionieren eines Arrays verwendet werden.

RESET entleert oder löscht alle Elemente (oder eine Dimension mit allen Unter-Dimensionen) einer oder mehrerer Variablen/Arrays. Wird bei RESET ein Variablenname angegeben, so wird diese Variable komplett geleert (aber nicht aus dem Speicher entfernt). Ist die Variable ein Array, so werden alle Elemente und alle Dimensionen dieses Arrays freigegeben und können neu zugewiesen werden.

#### Beispiele:

```
dec [a], [zahlen:'5']  
  
[a] = 'Ein Wert'  
[zahlen:'1'] = '5'  
[zahlen:'2'] = '13'  
[zahlen:'3'] = '21'  
[zahlen:'4'] = '79'  
[zahlen:'5'] = '2005'  
  
echo '[a] hat den Wert ' # [a]  
echo 'Das Array hat ' # [zahlen:'0'] # ' Einträge'  
  
reset [a]  
reset [zahlen]  
  
echo '[a] hat jetzt den Wert ' # [a]  
echo 'Das Array hat jetzt ' # [zahlen:'0'] # ' Einträge'  
end
```

#### Bemerkungen:

Es ist nicht mehr unbedingt notwendig, Variablen zu deklarieren, da diese automatisch deklariert werden sobald ihnen ein Wert zugewiesen wird. Die Verwendung von DEC ist allerdings in Prozeduren und bei der sauberen Script-Programmierung unerlässlich.

RESET dient rein zur manuellen Speicherverwaltung. Bei der Beendigung eines RapidBATCH-Scripts gibt RapidBATCH automatisch den reservierten Speicher wieder frei.

### IF ...ELSEIF...ELSE...ENDIF

#### Syntax:

```
IF Bedingung Anweisung  
  
IF Bedingung  
{Anweisungsblock}  
[ELSEIF Bedingung  
{Anweisungsblock}]  
[ELSE  
{Anweisungsblock}]  
ENDIF
```

**Beschreibung:**

Die IF-Anweisung ermöglicht es, Anweisungen bedingt auszuführen. Sie prüft, ob eine Bedingung das Ergebnis TRUE (wahr) zurückliefert. Ist dies der Fall, kann eine nachfolgende Anweisung oder ein so genannter Anweisungsblock ausgeführt werden. Soll nur eine Anweisung ausgeführt werden, so schreibt man diese direkt hinter die Bedingung, bei mehreren Anweisungen schreibt man den Anweisungsblock zwischen dem IF und einem ENDIF oder einem ELSE oder ELSEIF, wobei das ENDIF immer den gesamten IF-Block abschliesst. Wird ein ELSE angegeben, lassen sich alternativ Anweisungen ausführen, wenn die Bedingung des IFs das Ergebnis FALSE (falsch) liefert. Wird ein ELSEIF angegeben, lassen sich mehrere Bedingungen prüfen, wenn die erste Bedingung das Ergebnis FALSE liefert. Es können beliebig viele ELSEIF-Blöcke in einem IF-Block vorkommen, jedoch ist nur ein IF und ein ELSE je Block erlaubt. Anweisungsblöcke können weitere IF-Blöcke beinhalten (verschachtelte IF-Blöcke).

Eine Bedingung ist ein Vergleich zweier Werte, wobei diese Werte auf gleich (=), ungleich (!), grösser-als (>) und kleiner-als (<) geprüft werden können. Zudem sind die Operatoren grösser-gleich (>=) und kleiner-gleich (<=) möglich.

Um mehr als zwei Bedingungen bei einer IF-Anweisung zu überprüfen, lassen sich mehrere Bedingungen mit den logischen Operatoren UND (&) und ODER (|) verknüpfen, wobei der &-Operator nur TRUE zurückgibt, wenn alle mit diesem Operator verknüpften Bedingungen erfolgreich geprüft wurden. Beim |-Operator muss eine der beiden Bedingungen (oder auch beide) das Ergebnis TRUE liefern.

**Beispiele:**

```
if [a] = [b] echo 'A ist gleich B!'
```

```
if [a] >= [b] echo 'A ist größer oder gleich B!'
```

```
if [password] = 'swordfish'
    echo 'Login erfolgreich.'
else
    echo 'Falsches Passwort.'
    halt
endif
```

```
if [a] < '10'
    echo 'Fehler!'
    halt
elseif [b] <= '5'
    echo 'Alles in Ordnung!'
endif
```

```
rem Logisches UND, liefert nur wahr, wenn beide Variablen den Wert 10 enthalten.
[a] = '10'
[b] = [a]
if [a] = '10' & [b] = '10'
    echo 'A und B haben beide den Wert 10!'
endif
```

```
rem Logisches ODER, liefert nur wahr, wenn eine oder beide Variablen den Wert 10 enthalten.
[a] = '11'
[b] = '10'
if [a] = '10' | [b] = '10'
    echo 'A oder B hat den Wert 10!'
endif
```

```
rem Komplexeres Beispiel zur Anwendung von logischen Operatoren
if [a] = '10' & [b] = '10' | [a] = '9' & [b] = '12'
    echo 'Starte Taschenrechner...'
    call 'c:\windows\calc.exe', 'show'
endif
```

**Bemerkungen:**

IF-Blöcke können beliebig ineinander verschachtelt werden.

## REPEAT...CONT ... BREAK ... UNTIL

### Syntax:

```
REPEAT
  {Anweisungsblock}
  [CONT]
  {Anweisungsblock}
  [BREAK]
  {Anweisungsblock}
UNTIL Abbruchbedingung
```

### Beschreibung:

Mit REPEAT ... UNTIL lässt sich eine fussgesteuerte Schleife programmieren. Diese Möglichkeit gestattet es, bestimmte Anweisungen wiederholt auszuführen, bis die bei UNTIL gegebene Bedingung erfolgreich geprüft wird.

Eine Bedingung (in diesem Fall eine Abbruchbedingung) ist ein Vergleich zweier Werte, wobei diese Werte auf gleich (=), ungleich (!), grösser-als (>) und kleiner-als (<) geprüft werden können. Zudem sind die Operatoren größer-gleich (>=) und kleiner-gleich (<=) möglich.

Um mehr als zwei Bedingungen bei einer UNTIL-Anweisung zu überprüfen, lassen sich mehrere Bedingungen mit den logischen Operatoren UND (&) und ODER (!) verknüpfen, wobei der &-Operator nur TRUE zurückgibt, wenn alle mit diesem Operator verknüpften Bedingungen erfolgreich geprüft wurden. Beim !-Operator muss eine der beiden Bedingungen (oder auch beide) das Ergebnis TRUE liefern.

Wird die Abbruchbedingung nicht erreicht, endet die Ausführung des Scripts in einer Endlos-Schleife.

Um eine Schleife vorzeitig zu verlassen, verwenden Sie die Anweisung BREAK.

Die Anweisung CONT bewirkt, dass die Schleife ohne vorherige Prüfung der bei UNTIL angegebenen Bedingung wiederholt wird (CONT bewirkt damit einen Rücksprung zum REPEAT der Schleife).

### Beispiele:

```
[i] = 0
repeat
  [i] + '1'
  echo 'Durchlauf: ' # [i]
until [i] = '10'

rem Schleifenbedingung mit Bedingungsverknüpfung
[a] = '0'
[b] = '0'
repeat
  [a] + '1'
  [b] + '2'
until [a] > '10' & [b] > '20'

rem Schleife mit BREAK-Anweisung
[i] = 0
repeat
  [i] + '1'
  rem Verlassen der Schleife wenn [i] größer 5 ist
  if [i] > '5' break
  echo 'Durchlauf: ' # [i]
until [i] = '10'

rem Schleife mit CONT-Anweisung
[i] = 0
repeat
  [i] + '1'
  rem Wenn [i] den Wert 5 enthält, Schleife wiederholen
  if [i] = '5' cont
  echo 'Durchlauf: ' # [i]
until [i] = '10'
```

**Bemerkungen:**

REPEAT...UNTIL-Schleifen können beliebig ineinander verschachtelt werden.

**PROC...ENDPROC****Syntax:**

```
PROC Prozedurname [: Parameter-Liste]
  {Anweisungsblock}
  [RET]
  {Anweisungsblock}
ENDPROC
```

**Beschreibung:**

Definiert eine benutzerdefinierte Prozedur.

Prozeduren werden implementiert, um mehrere RapidBATCH-Anweisungen zu einem neuen Befehl zusammenzufassen, der im Script verwendet werden kann. Eine Prozedur kann einen beliebigen Namen sowie eine optionale und beliebige Anzahl an Parametervariablen besitzen, die beim Aufruf der Prozedur als lokale Prozedurvariablen behandelt werden. Mit Hilfe der Pointerfunktionalität können auch Zeiger auf Arrays und Variablen an die Prozedur als lokale Variablen übergeben und verarbeitet werden.

Mit der RET-Anweisung kann eine Prozedur vorzeitig beendet werden; Hingegen zu Funktionen ist bei in Prozeduren verwendeten RET-Anweisungen kein Rückgabewert erlaubt.

Eine Prozedur kann wiederum andere Prozeduren/Funktionen oder sich selbst rekursiv aufrufen. Funktions-/Prozedurdeklarationen in Prozeduren sind nicht erlaubt.

**Beispiele:**

```
rem Implementation der Prozedur "berechne"
proc berechne: [a], [b]
  dec [text]
  [text] # [a] # ' + ' # [b] # ' = ' # ([a] + [b]) # [new_line]
  [text] # [a] # ' - ' # [b] # ' = ' # ([a] - [b]) # [new_line]
  [text] # [a] # ' * ' # [b] # ' = ' # ([a] * [b]) # [new_line]
  [text] # [a] # ' / ' # [b] # ' = ' # ([a] / [b]) # [new_line]
  [text] # [a] # ' ~ ' # [b] # ' = ' # ([a] ~ [b])
  echo [text]
endproc

rem Aufruf der Prozedur "berechne"
berechne '10', '5'

rem Implementation der Prozedur "ausgabe_array" mit Übergabe eines Array-Pointers
proc ausgabe_array: [*array]
  dec [i],[text]
  if [array:'0'] = '0' ret
  repeat
    [i] + '1'
    [text] # 'Index ' # [i] # ': ' # [array[i]] # [new_line]
  until [i] = [array'0']
  echo [text]
endproc

[test:'1'] = 'Array Wert 1'
[test:'2'] = 'Array Wert 2'
[test:'3'] = 'Array Wert 3'
[test:'4'] = 'Array Wert 4'

rem Aufruf der Prozedur "ausgabe_array"
ausgabe_array [test]
end
```

**Bemerkungen:**

Um eine Funktion zu implementieren, verwenden Sie FUNC...ENDFUNC.

Mehrere Prozeduren und Funktionen können als Bibliothek in eine Script-Datei implementiert werden, die dann mit Hilfe der INCLUDE-Anweisung in andere RapidBATCH-Scripts, die diese Prozeduren verwenden, eingebunden wird.

Eine Prozedur kann nicht vor ihrer Definition aufgerufen werden.

RET kann nicht ausserhalb einer Prozedur oder Funktion verwendet werden.

**FUNC...ENDFUNC****Syntax:**

```
FUNC Funktionsname [:Parameter-Liste]
    {Anweisungsblock}
    [RET [Rückgabewert] ]
    {Anweisungsblock}
ENDFUNC
```

**Beschreibung:**

Definiert eine benutzerdefinierte Funktion.

Funktionen werden implementiert, um mehrere RapidBATCH-Anweisungen zu einer neuen Funktion zusammenzufassen, die im Script verwendet werden kann. Eine Funktion kann einen beliebigen Namen sowie eine optionale und beliebige Anzahl an Parametervariablen besitzen, die beim Aufruf der Funktion als lokale Funktionsvariablen behandelt werden. Mit Hilfe der Pointerfunktionalität können auch Zeiger auf Arrays und Variablen an die Funktion als lokale Variablen übergeben und verarbeitet werden.

Funktionen haben Grundsätzlich einen Rückgabewert, d.h. beim Aufruf einer benutzerdefinierten Funktion muss eine Rückgabevariable (Return-Variable) äquivalent Funktionsaufrufen von RapidBATCH build-in Funktionen angegeben werden.

Um einen Wert zurückzugeben, verwenden Sie die RET-Anweisung, die als Parameter den zurückzugebenden Wert erwartet. Wird in einer Funktion eine RET-Anweisung ohne Parameter verwendet, so gibt die Funktion einen Leerstring an die Rückgabevariable zurück.

Eine Funktion kann wiederum andere Funktionen/Prozeduren oder sich selbst rekursiv aufrufen. Funktions-/Prozedurdeklarationen in Funktionen sind nicht erlaubt.

**Beispiele:**

```
rem Definition der Funktion "quadrat"
func quadrat: [wert]
    ret [wert] * [wert]
endfunc

rem Aufruf der Funktion "quadrat"
quadrat [ergebnis] = '4'
echo 'Ergebnis: ' # [ergebnis]

rem Definition der Funktion "fakultaet"
func fakultaet: [x]
    dec [res]

    if [x] = '0'
        [res] = '1'
    else
        rem Rekursiver Aufruf von "fakultaet"
        fakultaet [res] = [x] - '1'
        [res] * [x]
    endif

    ret [res]
endfunc
```

```
rem Aufruf der Funktion "fakultaet"
fakultaet [ergebnis] = '4'

echo 'Die Fakultät von 4 ist ' # [ergebnis]
end
```

**Bemerkungen:**

Um eine Prozedur zu definieren, verwenden Sie PROC...ENDPROC.

Mehrere Prozeduren und Funktionen können als Bibliothek in eine Script-Datei implementiert werden, die dann mit Hilfe der INCLUDE-Anweisung in andere RapidBATCH-Scripts, die diese Prozeduren/Funktionen verwenden, eingebunden wird.

Eine Funktion kann nicht vor ihrer Definition aufgerufen werden.

RET kann nicht ausserhalb einer Prozedur oder Funktion verwendet werden.

## GOTO

**Syntax:**

```
GOTO label
.
.
.
%label
```

**Beschreibung:**

GOTO springt zu einem dort spezifizierten Label im Script. Dies ermöglicht äusserst komplexe Programme, da Anweisungen wiederholt ausgeführt oder übersprungen werden können. Ein Label ist durch ein vorangestelltes %-Zeichen im Code gekennzeichnet.

Seit RapidBATCH 5.0 ist es auch möglich, variable Werte an GOTO zu übergeben, d.h. der Name des Labels, welches angesprungen werden soll, kann eine Variable oder auch ein aus variablen und statischen Werten zusammengesetzter Label-Name sein.

**Beispiele:**

```
include 'dialog.rb'

rem Einfaches, statisches GOTO
echo 'Willkommen im GOTO-Testscript!'
goto 'weiter1'

echo 'Diese Meldung wird niemals ausgegeben :('

%weiter1

rem Ein dynamisches GOTO
btnmenu [geschwindigkeit] = 'Wählen Sie eine Geschwindigkeit!', '30 km/h|60 km/h|120 km/h'

goto [geschwindigkeit]

%30 km/h
echo 'Das ist aber sehr langsam!'
goto 'weiter2'

%60 km/h
echo 'Das ist ganz okay!'
goto 'weiter2'

%120 km/h
echo 'SEHR SCHNELL! Wollen Sie ein Rennen fahren?'
```

```
%weiter2  
echo 'Weiterhin gute Fahrt mit Ihren ' # [geschwindigkeit] # ' ...'
```

**Bemerkungen:**

Der Name des Labels muss bei GOTO ohne das "%" -Zeichen angegeben werden!

Es ist nicht erlaubt, mit einem GOTO-Sprung aus einer oder in eine Prozedur zu springen. Ist dies der Fall, bricht die Scriptausführung mit einer entsprechenden Fehlermeldung ab. GOTO-Sprünge in andere Include-Dateien sind möglich. Sprünge können in alle Richtungen (ob vorwärts oder rückwärts) im Script durchgeführt werden.

Verwenden Sie die GOSUB-Anweisung, um eine Sprungmarke als Unterprozedur aufzurufen.

## GOSUB, RESUME

**Syntax:**

```
GOSUB label  
.  
.  
.  
%label.  
.  
.  
RESUME
```

**Beschreibung:**

GOSUB arbeitet ähnlich wie GOTO. Hier wird zu einem Label gesprungen, welches hier als Unterprogramm (Sub) angesehen wird. Sobald eine RESUME-Anweisung auftritt, wird das Script hinter dem letzten GOSUB-Aufruf fortgesetzt. Dies ermöglicht äusserst komplexe Programmkonstruktionen. GOSUB...RESUME-Konstruktionen lassen sich beliebig ineinander verschachteln (z.B. lassen sich aus Subs wiederum andere GOSUB-Sprünge durchführen).

Seit RapidBATCH 5.0 ist es auch möglich, variable Werte an GOSUB zu übergeben, d.h. der Name des Labels, welches angesprungen werden soll, kann eine Variable oder auch ein aus variablen und statischen Werten zusammengesetzter Label-Name sein.

**Beispiele:**

```
include 'dialog.rb'  
  
rem Ein statischer GOSUB-Aufruf  
[i] = '0'  
repeat  
    [i] + '1'  
    gosub 'square'  
until [i] = '10'  
  
rem Auf zum nächsten Beispiel...  
goto 'weiter'  
  
%square  
    echo [i] # ' * ' # [i] # ' = ' # ([i] * [i])  
    resume  
  
%weiter  
rem Ein dynamischer GOSUB-Aufruf mit einer Variablen  
rem (Zur Laufzeit sieht dieses Beispiel genauso aus wie das von GOTO,  
rem allerdings ist die Umsetzung hier komplett anders)  
  
btnmenu [geschwindigkeit] = 'Wählen Sie eine Geschwindigkeit!', '30 km/h|60 km/h|120 km/h'  
  
gosub [geschwindigkeit]  
  
echo 'Weiterhin gute Fahrt mit Ihren ' # [geschwindigkeit] # ' ...'
```



## halt

```
%30 km/h
echo 'Das ist aber sehr langsam!'
resume

%60 km/h
echo 'Das ist ganz okay!'
resume

%120 km/h
echo 'SEHR SCHNELL! Wollen Sie ein Rennen fahren?'
resume
```

### Bemerkungen:

Ein RESUME muss nicht gegeben sein. In solch einem Fall ist GOSUB äquivalent zu GOTO. Wird ein RESUME ohne ein vorheriges GOSUB angegeben, wird dieses ignoriert und die Scriptausführung hinter RESUME fortgesetzt.

In RapidBATCH 5 wird von GOSUB...RESUME-Konstruktionen abgeraten, da man hier Zugriff auf die weitaus professionelleren Möglichkeiten der modernen prozeduralen Programmierung mit Hilfe der Anweisungen PROC...ENDPROC oder FUNC...ENDFUNC hat.

GOSUB...RESUME-Konstruktionen können aber bei der Implementierung von Ereignis-Routinen von eigenen Dialogen sehr von Nutzen sein.

## HALT

### Syntax:

HALT

### Beschreibung:

Beendet das Script sofort und gibt alle verwendeten Speicher- und Systemressourcen frei.

### Beispiele:

```
chdir 'C:\Test'
if [errorcode] = '-1'
    echo 'C:\Test existiert nicht!'
    rem Script beenden
    halt
endif
```

### Bemerkungen:

keine

## INCLUDE, INCLUDE\_ONCE

### Syntax:

INCLUDE Includedatei  
INCLUDE\_ONCE Includedatei

### Beschreibung:

INCLUDE ermöglicht die Importierung einer RapidBATCH-Scriptdatei in ein RapidBATCH-Script. Diese inkludierte Datei wird komplett an der Stelle, wo das INCLUDE steht, ausgeführt, d.h. es werden entsprechende Funktionen verfügbar gemacht und Variablen zum aktuell ausgeführten Script hinzudeklariert.

Sinn und Zweck von Include-Dateien ist die Aufteilung eines Projektes in mehrere kleine Dateien und die Implementierung von Prozedur- und Funktionsbibliotheken. Man kann daher einmal eine Bibliothek schreiben und alle Funktionen und Prozeduren dieser Bibliothek mittels INCLUDE in jedem Script, wo die Funktionen gebraucht werden, verfügbar machen.

INCLUDE\_ONCE hat denselben Zweck wie INCLUDE, nur wird hier das zu inkludierende Script wirklich nur ein mal ausgeführt. Kommt der Interpreter/Compiler ein weiteres Mal über solch eine Zeile mit selbem Dateinamen, so wird das Script nicht noch einmal importiert.

**Beispiele:**

```
rem Inkludieren der Array-Bibliothek
include 'array.rb'

rem Array füllen
[test:'1'] = 'Hello World'
[test:'2'] = 'Hello World'
[test:'3'] = 'Hello World'

rem Aufruf der Bibliotheksprozedur "viewarray"
viewarray [test]
end
```

**Bemerkungen:**

Als Dateiname kann keine Variable mit dem Dateinamen als Inhalt angegeben werden.

Wenn bei INCLUDE oder INCLUDE\_ONCE ein relativer Pfad angegeben wurde, der vom Scriptverzeichnis aus nicht existiert, sucht RapidBATCH automatisch im Library-Verzeichnis (im RapidBATCH Installationsverzeichnis das Unterverzeichnis \LIB). Darin befinden sich einige Standardbibliotheken, die RapidBATCH als erweiterte Sprachbasis mitführt. Durch Kopieren einer Library in dieses Verzeichnis kann sie von allen Scripts auf dem System durch einfache Angabe des Dateinamens (ohne absoluten Pfad) importiert werden.

Wenn Sie automatisch bei jedem Script, welches von RapidBATCH ausgeführt wird, bereits vorab einige Libraries inkludieren möchten, tragen Sie einfach den entsprechenden Sourcecode in die Datei default.rb im \LIB-Verzeichnis ein. Diese Datei wird vom Interpreter und vom Compiler automatisch für jedes Script inkludiert, ohne einen expliziten Aufruf von INCLUDE oder INCLUDE\_ONCE.

## REM

**Syntax:**

REM Kommentar

**Beschreibung:**

REM markiert eine Kommentarzeile im Code, mit der das Script kommentiert werden kann. REM führt keine Aktion aus, sie teilt dem Compiler/Interpreter lediglich mit, dass diese Zeile ignoriert werden soll.

**Beispiele:**

```
rem Ausgabe des Textes "Hello World"
echo 'Hello World'
end
```

**Bemerkungen:**

Im mitgelieferten Scripteditor "RapidBATCH Builder" können größere Codeblöcke schnell und einfach mit der Tastenkombination Strg+R aus- und einkommentiert werden.

## WAIT

**Syntax:**

WAIT Millisekunden

**Beschreibung:**

Unterbricht die Scriptausführung für die angegebene Zeit an Millisekunden. 1000 Millisekunden entsprechen einer Sekunde.

**Beispiele:**

```
shell 'C:\windows\calc.exe', 'show'
rem Zwei Sekunden warten...
wait '2000'
echo 'Viel Spass mit dem Calculator ;)'
end
```

**Bemerkungen:**

keine

**EXT****Syntax:**

EXT Rückgabewert = DLL-Datei, Funktionsname [, Parameter1 [ ..., Parametern] ]

**Beschreibung:**

Ruft eine externe Funktion aus einer DLL-Bibliothekdatei auf.

Mit EXT können daher selbst-entwickelte und spezielle Funktionalitäten in RapidBATCH verwendet und genutzt werden. Diese Funktionen werden in Programmiersprachen wie C, C++, Pascal oder Delphi realisiert.

Über EXT aufgerufene Funktionen müssen dem EXT-Schnittstellenformat entsprechen, ansonsten sind sie nicht korrekt aufrufbar. Das Format für eine Funktion (hier in der Programmiersprache C realisiert) ist wie Folgt definiert:

```
int __declspec( dllexport ) _cdecl
    Funktionsname( char** output, char** input, int input_count )
{
    /*
    Hier folgt der ausführbare Code ...
    */

    return 0;
}
```

Die Parameter, die einer externen Funktion übergeben werden, werden wie folgt gehandhabt:

- char\*\* output
- Hierbei handelt es sich um einen Pointer, der entsprechend mit Speicher allokiert werden muß, wenn die Funktion einen Wert an die beim Aufruf von EXT übergebene Rückgabeveriable zurückgeben soll. Dieser Speicher sollte mit der Windows API Funktion LocalAlloc() und dem Parameter LPTR allokiert werden. Wird \*output auf (char\*)NULL gesetzt bzw. belassen, gibt EXT einen Leerstring zurück. Die Schnittstelle gibt den mit LocalAlloc() allokierten Wert automatisch wieder frei.
- char\*\* input
- Bei diesem Pointer handelt es sich um ein Array von Char-Pointern, in der jeder Eintrag einem Parameterwert entspricht, der beim Aufruf von EXT im RapidBATCH-Script übergeben wurde.
- int input\_count
- Enthält die Anzahl der Einträge im Array input[].

## Beispiele:

```
rem Funktionen der DLL-Bibliothek (C-Code)
rem
rem     #define BUFFSIZE 10000
rem     #define TEXTSIZE 100
rem     #define ERROR_BAD_INPUT 11
rem
rem     char*      gsSeparator      = (char*)NULL;
rem     char*      gsBuffer        = (char*)NULL;
rem
rem     BOOL CALLBACK EnumWindowsProc( HWND hwnd, LPARAM lParam )
rem     {
rem         char      Title[TEXTSIZE];
rem         memset( Title, '\0', TEXTSIZE );
rem
rem         GetWindowText( hwnd, Title, TEXTSIZE-1 );
rem
rem         if( *Title != '\0' )
rem         {
rem             if( IsWindowVisible( hwnd ) )
rem             {
rem                 if( *gsBuffer != '\0' )
rem                     strcat( gsBuffer, gsSeparator );
rem
rem                 strcat( gsBuffer, Title );
rem
rem                 if( strlen( gsBuffer ) + strlen( gsSeparator ) + TEXTSIZE < BUFFSIZE )
rem                     return( TRUE );
rem                 else
rem                     return( FALSE );
rem             }
rem         }
rem
rem         return( TRUE );
rem     }
rem
rem     int __declspec(dllexport ) _cdecl rbxCreateWindowList( char** Output, char** Input, int InputCount )
rem     {
rem         if( Output == NULL )
rem             return( -1 );
rem
rem         *Output = LocalAlloc( LPTR, BUFFSIZE );
rem         if( *Output == NULL )
rem             return( -1 );
rem
rem         if( InputCount != 1 || Input == NULL || *Input == NULL || **Input == 0 ||
rem             strlen( *Input ) > TEXTSIZE )
rem         {
rem             return( -2 );
rem         }
rem
rem         gsBuffer = *Output;
rem         gsSeparator = *Input;
rem
rem         EnumWindows( EnumWindowsProc, ( LPARAM )NULL );
rem
rem         return( 0 );
rem     }
rem
rem Aufruf der Funktion:
ext [fensterliste] = 'Test.dll', 'rbxCreateWindowList', [new_line]
echo 'Offene Fenster: ' # [new_line] # [new_line] # [fensterliste]
end
```

**Bemerkungen:**

Beim ersten Aufruf einer Funktion aus einer DLL wird die DLL in den Speicher geladen und initialisiert. Die DLL bleibt dann bis zum Programmende im Speicher und wird dann zusammen mit der Anwendung automatisch entladen.

Jede DLL muss eine Funktion LibMain() enthalten. Diese Funktion wird automatisch beim Laden und Entladen der DLL aufgerufen. Die DLL kann in dieser Routine z.B. bei Laden Variablen initialisieren oder beim Beenden benutzte Objekte wieder freigeben oder temporäre Dateien löschen. Die Funktion LibMain() muss beim Laden der DLL den Wert TRUE zurückgeben, sonst wird der Ladevorgang abgebrochen und die DLL wird sofort wieder entladen. RapidBATCH würde dann [errorcode] = '-1' liefern - so als wenn die DLL nicht gefunden wurde. Beim Entladen wird der Rückgabewert der Funktion LibMain ignoriert.

Beim ersten Aufruf eines EXT-Befehls wird zuerst die DLL geladen (und damit LibMain() ausgeführt), dann wird die Funktion gesucht und aufgerufen. Bei weiteren EXT-Befehlen, die dieselbe DLL benutzen, wird direkt die angegebene Funktion gesucht und aufgerufen.

## Build-in Funktionen

### GETCHR, GETASC

**Syntax:**

GETCHR Zeichen = ASCII-Wert  
GETASC ASCII-Wert = Zeichen

**Beschreibung:**

GETCHR gibt das durch den numerischen ASCII-Code spezifizierte ASCII-Zeichen zurück.  
GETASC liefert zu dem ersten Zeichen eines Stringausdrucks den dazugehörigen ASCII-Code zurück.

**Beispiele:**

```
getchr [zeichen] = '65'  
echo 'Geholtes Zeichen: ' # [zeichen]  
getasc [ascii] = 'B'  
echo 'Das Zeichen "B" hat den ASCII-Code ' # [ascii]  
end
```

**Bemerkungen:**

keine

### GETLEN

**Syntax:**

GETLEN Länge = Ausdruck

**Beschreibung:**

Gibt die Anzahl der Zeichen (= Länge) des übergebenen Stringausdrucks zurück.

**Beispiele:**

```
getlen [len] = 'Hello World'  
echo 'Der Text "Hello World" ist ' # [len] # ' Zeichen lang.'  
end
```

**Bemerkungen:**

keine

### GETPOS

**Syntax:**

GETPOS Position = Stringausdruck, Suchstring, Anzahl

**Beschreibung:**

Sucht den angegebenen Suchstring im Stringausdruck und gibt dessen Position im String zurück.  
Mit Hilfe des Anzahl-Parameters kann auch nach dem zweiten, dritten usw. Aufkommen des Suchstrings im Stringausdruck gesucht werden.  
Wird der Suchstring nicht an der angegebenen Position mit der gegebenen Anzahl gefunden, ist der Rückgabewert '0'.

**Beispiele:**

```
getpos [pos] = 'Hello World', 'llo', '1'  
echo 'Position ist: ' # [pos]  
  
getpos [pos] = 'Hello World', 'l', '3'  
echo 'Position ist: ' # [pos]  
end
```

**Bemerkungen:**

Hat die Variable [case\_sensitivity] den Wert '-1', so berücksichtigt GETPOS auch Stringmuster, die sich von der Groß-/Kleinschreibung her von der Schreibweise des gegebenen Suchstrings unterscheiden. D.h. wenn der Suchstring z.B. in Kleinschreibweise vorliegt und [case\_sensitivity] den Wert '-1' (false) hat, so wird auch ein in Großbuchstaben geschriebener, äquivalenter Substring gefunden.

**GETTOK****Syntax:**

GETTOK Token = Stringausdruck, Trennstring, Token-Zähler

**Beschreibung:**

Die Tokenizer-Funktion GETTOK liefert ein so genanntes Token aus einem Stringausdruck zurück. Ein Token ist ein Substring, der durch ein Trennzeichen, z.B. das |-Zeichen von anderen Tokens getrennt ist. Als Trennzeichen (Trennstring) kann auch ein Stringausdruck angegeben werden, der mehr als 1 Zeichen enthält. GETTOK liefert als Rückgabewert das Token von der übergebenen Token-Position zurück.

**Beispiele:**

```
findfiles [files] = 'c:\jmksf\rb5\*.exe', '1'

[count] = '1'
repeat
    gettok [tok] = [files], '|', [count]
    if [tok] != ''
        echo 'Token an Position ' # [count] # ' ist: ' # [tok]
    endif
    [count] + '1'
until [tok] = ''
end
```

**Bemerkungen:**

Wird das Ende des Strings erreicht, gibt GETTOK das bis dahin gelesene Token zurück (sofern das letzte Zeichen kein Token-Trenner ist, wobei dies dann auch keine Rolle spielt).

Hat die Variable [case\_sensitivity] den Wert '-1', so berücksichtigt GETTOK auch Token-Separatoren als solche, wenn diese sich von Groß-/Kleinschreibung her von der Schreibweise des gegebenen String-Separators unterscheiden. D.h. wenn der Token-Separator in Kleinschreibung vorliegt und [case\_sensitivity] den Wert '-1' (false) hat, so wird auch ein großgeschriebener Token-Separator als solcher erkannt.

**GETCHARAT****Syntax:**

GETCHARAT Zeichen = Stringausdruck, Zeichenposition

**Beschreibung:**

Liefert das Zeichen an der gegebenen Position im gegebenen Stringausdruck zurück.

**Beispiele:**

```
[text] = 'Hello World'
rem      12345678901
getcharat [chr] = [text], '5'
echo 'Das Zeichen an Position 5 im String ' # [text] # ' ist ein "' # [chr] # "'"
end
```

**Bemerkungen:**

keine



## UPVAR, LOWVAR

### Syntax:

UPVAR Ergebnis = Ausdruck  
LOWVAR Ergebnis = Ausdruck

### Beschreibung:

UPVAR wandelt alle Zeichen eines Stringausdrucks in Grossbuchstaben um.  
LOWVAR wandelt alle Zeichen eines Stringausdrucks in Kleinbuchstaben um.

Bereits gross- bzw. kleingeschriebene Zeichen werden von der jeweiligen Funktion ignoriert.

### Beispiele:

```
[text] = 'Ein Test'
upvar [utext] = [text]
lowvar [ltext] = [text]
echo [text] # ' in Großbuchstaben: ' # [utext]
echo [text] # ' in Kleinbuchstaben: ' # [ltext]
end
```

### Bemerkungen:

keine

## TRIMVAR

### Syntax:

TRIMVAR Ergebnis = Ausdruck

### Beschreibung:

Entfernt führende und nachfolgende Leerzeichen aus einer Zeichenfolge und gibt diese zurück.

### Beispiele:

```
[text] = '   RapidBATCH   '
echo '***' # [text] # '***'
trimvar [trim_text] = [text]
echo '***' # [trim_text] # '***'
end
```

### Bemerkungen:

keine

## COPYVAR

### Syntax:

COPYVAR Teilstring = Stringausdruck, Start, Länge

### Beschreibung:

COPYVAR gibt einen Teilstring aus einem String zurück

Der Start-Parameter definiert dabei die Position des ersten Zeichens, ab dem der Teilstring kopiert werden soll, der Länge-Parameter die Anzahl der Zeichen ab der Startposition, die in den Teilstring kopiert werden sollen.

### Beispiele:

```
[name] = 'Arnold Schwarzenegger'
rem      123456789012345678901 (Zeichenpositionen)
copyvar [nachname] = [name], '8', '14'
copyvar [vorname] = [name], '1', '6'
```

```
echo [nachname] # ' , ' # [vorname]
end
```

**Bemerkungen:**

Wenn Sie als Start-Parameter den Wert '-1' übergeben, werden Länge-Zeichen von rechts aus dem String gelesen und als Teilstring zurückgegeben.

## CNTVAR

**Syntax:**

CNTVAR Anzahl = Stringausdruck, Suchstring

**Beschreibung:**

Gibt die Anzahl der im Stringausdruck enthaltenen Suchstring-Ausdrücke zurück.

**Beispiele:**

```
readfile [text] = 'C:\jmksf\rb5\readme.txt', '0'

cntvar [anzahl] = [text], 'RapidBATCH'
echo 'Die Datei beinhaltet ' # [anzahl] # ' X das Wort "RapidBATCH"'
end
```

**Bemerkungen:**

Hat die Variable [case\_sensitivity] den Wert '-1', so zählt CNTVAR auch Stringmuster, die sich von der Groß-/Kleinschreibung her von der Schreibweise des gegebenen Suchstrings unterscheiden. D.h. wenn der Suchstring z.B. in Kleinschreibweise vorliegt, und [case\_sensitivity] den Wert '-1' (false) hat, so wird auch ein in Großbuchstaben geschriebener, äquivalenter Substring mitgezählt.

## REPLACEVAR

**Syntax:**

REPLACEVAR Ergebnisstring = Stringausdruck, Suchstring, Ersetzstring

**Beschreibung:**

Ersetzt jedes Aufkommen des gegebenen Suchstrings durch den Ersetzstring im gegebenen Stringausdruck. Rückgabewert ist der Stringausdruck mit den ersetzten Werten.

**Beispiele:**

```
chdir 'c:\jmksf\rb5'
findfiles [files] = '*.exe', '1'
replacevar [files] = [files], '|', [new_line]
echo 'Inhalt von ' # [current] # [new_line] # [new_line] # [files]
end
```

**Bemerkungen:**

Hat die Variable [case\_sensitivity] den Wert '-1', so berücksichtigt REPLACEVAR auch Stringmuster, die sich von der Groß-/Kleinschreibung her von der Schreibweise des gegebenen Suchstrings unterscheiden. D.h. wenn der Suchstring z.B. in Kleinschreibweise vorliegt und [case\_sensitivity] den Wert '-1' (false) hat, so wird auch ein in Großbuchstaben geschriebener, äquivalenter Substring als solcher erkannt und ersetzt.

## RANDVAR

### Syntax:

RANDVAR Zufallszahl = Maximalwert

### Beschreibung:

Erzeugt eine Pseudo-Zufallszahl, die zwischen 0 und dem übergebenen Maximalwert - 1 liegt. Wenn der Funktion beispielsweise 40 übergeben wird, ist der mögliche Zufallswert zwischen 0 und 40.

### Beispiele:

```
rem Erzeuge Zufallszahl...
randvar [random] = '255'
echo 'Zufallszahl: ' # [random]
end
```

### Bemerkungen:

Bei der Implementation von RANDVAR wurde darauf geachtet, dass sich Zahlen nicht wiederholen. Es kann aber durchaus vorkommen, dass ein bestimmtes Muster bei mehrfachen Generierungen entsteht, da es sich bei RANDVAR um einen Pseudo-Zufallszahlengenerator handelt. Als Basis für die Zufallszahlen verwendet RANDVAR die Systemuhrzeit.

## Build-in Dialoge

### ECHO

#### Syntax:

ECHO Meldungstext

#### Beschreibung:

Zeigt ein einfaches Meldungsfenster mit OK-Button an.



#### Beispiele:

```
echo 'Hello World'

[a] = '5' + '4'
echo 'Ergebnis ist: ' # [a]
end
```

#### Bemerkungen:

Der Standard-Text für die Titelzeile von ECHO-Meldungsfenstern kann über die Variable [Echo\_Title] geändert werden. Verwenden Sie die MSGBOX-Anweisung, um ein individuell gestaltetes Messagebox-Fenster bereitzustellen.

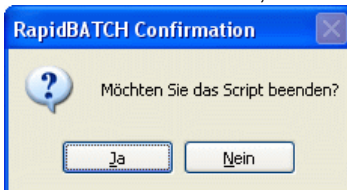
### CONFIRM

#### Syntax:

CONFIRM Auswahl = Meldungstext

#### Beschreibung:

Die CONFIRM-Funktion ist eine Abkürzung für ein Meldungsfenster mit JA-NEIN-Option. Drückt der Benutzer JA, so gibt die Funktion den Wert 0, andernfalls -1 zurück.



#### Beispiele:

```
repeat
    confirm [quit] = 'Möchten Sie das Script beenden?'
until [quit] = [true]
echo 'Script beendet.'
end
```

#### Bemerkungen:

Der Standard-Text für die Titelzeile von CONFIRM-Meldungsfenstern kann über die Variable [Confirm\_Title] geändert werden.

Verwenden Sie die MSGBOX-Anweisung, um ein individuell gestaltetes Messagebox-Fenster bereitzustellen.

## MSGBOX

### Syntax:

MSGBOX Titel, Meldungstext, Style  
MSGBOX Ergebnis = Titel, Meldungstext, Style

### Beschreibung:


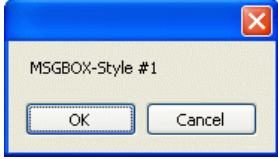
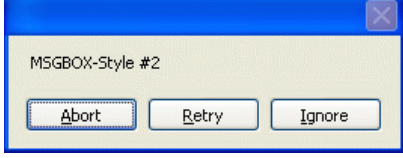
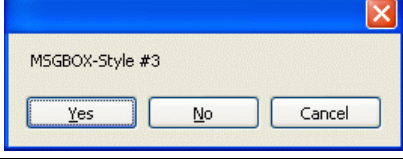
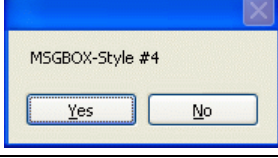
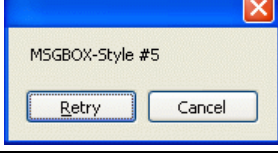
Zeigt ein individuelles Meldungsfenster an.

MSGBOX kann sowohl als Funktion als auch als Anweisung verwendet werden.






Die Anweisung/Funktion erwartet einen Titel, einen Meldungstext sowie einen so genannten Meldungsfenster-Style, welcher entweder durch eine Zahl (was äusserst individuelle Möglichkeiten zulässt) oder durch eine der build-in Konstanten, wie z.B. OK oder YESNOCANCEL repräsentiert wird.

Welcher der Buttons gedrückt wurde wird an die Variable [errorcode] oder im Falle eines Funktionsaufrufes von MSGBOX an die Returnvariable übergeben.

Die Rückgabewerte der jeweiligen Buttons sowie die Buttontypen sind wie folgt definiert:

Style-ID:	Beispiel	Button-Style (jew. Rückgabewert an [errorcode] bzw. Returnvariable in Klammern):
0		OK (1)
1		OK (1) + CANCEL (2)
2		CANCEL (3) + RETRY (4) + IGNORE (5)
3		YES (6) + NO (7) + CANCEL (2)
4		YES (6) + NO (7)
5		RETRY (4) + CANCEL (2)

Durch Addition eines der folgenden Icon-Styles kann auch ein individuelles Icon angezeigt werden.

Icon-Style:	Beispiel	Icon-Style:	Beispiel	Icon-Style:	Beispiel
0		16		32	
48		64			

## Beispiele:

```
rem Einfaches Meldungsfenster (MSGBOX-Anweisung)
msgbox 'Hallo!', 'Herzlich Willkommen!', '64'

rem Meldungsfenster mit Ja-Nein-Abbrechen-Option (MSGBOX-Funktion)
msgbox [antwort] = 'Script beenden?', 'Möchten Sie die Datei überschreiben', 'yesnocancel'
if [antwort] = '6'
    echo 'Sie haben JA gewählt.'
elseif [antwort] = '7'
    echo 'Sie haben NEIN gewählt.'
else
    echo 'Sie haben ABBRECHEN gewählt.'
endif
```

## Bemerkungen:

In RapidBATCH 2.0, 2.2 und 3.0 hiess diese Anweisung noch ECHOBOX. Alternativ kann auch weiterhin ECHOBOX benutzt werden, MSGBOX ist jedoch die moderne Variante ab RapidBATCH 4.0.

## INPUTBOX, PWDBOX

### Syntax:

INPUTBOX Eingabe = Titel, Eingabeaufforderung, Default-Wert  
PWDBOX Eingabe = Titel, Eingabeaufforderung, Default-Wert

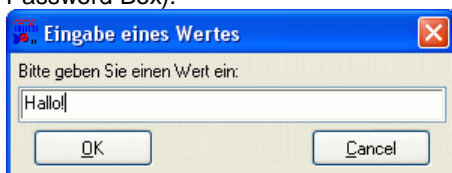
### Beschreibung:

INPUTBOX zeigt einen einfachen Dialog zur Eingabe eines Wertes an.

Die Funktion erwartet einen Dialogtitel, einen Eingabeaufforderungstext sowie einen Default-Wert, der im Eingabefeld vorgeblendet wird.

Wenn der OK-Button im Dialog gedrückt wurde, ist der Rückgabewert der Wert bzw. Text, der im Eingabefeld eingegeben wurde. Wird der CANCEL-Button gedrückt, so liefert die Funktion einen Leerstring zurück.

PWDBOX ist eine Abwandlung von INPUTBOX zur maskierten Eingabe von Werten, z.B. Passwörtern (PWDBOX = Password-Box).



### Beispiele:

```
rem Einfache Abfrage mittels INPUTBOX
```

```
inputbox [name] = 'Dein Name', 'Wie heißt du?', 'Anonymous'
echo 'Hallo, ' # [name]

rem Passwortabfrage mittels PWDBOX
pwdbox [passwort] = 'Passwort', 'Bitte geben Sie Ihr Passwort ein:', ''
if [passwort] = ''
    echo 'Fehler, kein Passwort eingegeben.'
else
    echo 'Dankeschön, Ihr Passwort lautet: ' # [passwort]
endif
```

## Bemerkungen:

Die Beschriftung der Buttons OK und CANCEL kann über die Variablen [InputBox\_Ok] und [InputBox\_Cancel] gesetzt werden. Dies trifft dann auch auf die PWDBOX zu, da diese den selben Dialog, allerdings mit einem anderen Eingabefeld, verwendet.

Wird die Beschriftung der Buttons auf einen Leerstring gesetzt, wird der Button aus dem Dialog ausgeblendet.

Die Position des Dialogs kann mit Hilfe der Variablen [InputBox\_X] und [InputBox\_Y] verändert werden. Hat [InputBox\_X] und/oder [InputBox\_Y] den Wert 0 wird der Dialog automatisch jeweils horizontal oder vertikal zentriert (dies ist die Standard-Einstellung).

Die Angabe bei Positionierungswerten erfolgt jeweils in Pixeln.

Die Größe von INPUTBOX/PWDBOX ist prinzipiell nicht veränderbar.

Das Dialog-Icon kann global für alle Dialoge über die Variable [app\_icon] auf eine externe .ICO-Datei gesetzt werden.

## EDITBOX

### Syntax:

EDITBOX Eingabe = Titel, Text, Style

### Beschreibung:

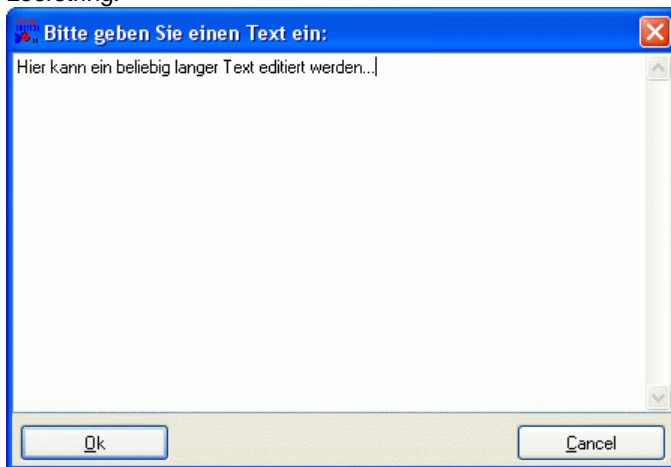
EDITBOX stellt einen Dialog zur Verfügung, in dem mehrzeilige Texte editiert und betrachtet werden können.

Die Funktion erwartet einen Dialogtitel, einen im Editierfeld vorgegebenen Text und einen Anzeigemodus.

Über den Anzeigemodus lässt sich bestimmen, ob der Text im Editierfeld geändert oder nur angesehen werden kann.

Wird der Wert '0' oder die äquivalente Konstante 'writeable' angegeben, kann der Benutzer den Text editieren; wird der Wert '-1' oder 'readonly' angegeben, so kann der Benutzer den Text nicht verändern.

Wird im Dialog der OK-Button gedrückt, so liefert die Funktion den Text aus dem Eingabefeld zurück, ansonsten einen Leerstring.





## Beispiele:

```
rem Einfache, mehrzeilige Texteingabe
editbox [text] = 'Texteditor', '', 'writeable'
echo [text]

rem Verwendung der EDITBOX zur Textbetrachtung (readonly-Modus)
editbox [text] = 'Textanzeige', 'Dieser Text kann nicht editiert werden!', 'readonly'
echo [text]
end
```

## Bemerkungen:

Die Beschriftung der Buttons OK und CANCEL kann über die Variablen [EditBox\_Ok] und [EditBox\_Cancel] gesetzt werden. Wird die Beschriftung eines Buttons auf einen Leerstring gesetzt, wird der Button aus dem Dialog ausgeblendet.

Die Position des Dialogs kann mit Hilfe der Variablen [EditBox\_X] und [EditBox\_Y] verändert werden. Hat [EditBox\_X] und/oder [EditBox\_Y] den Wert 0 wird der Dialog automatisch jeweils horizontal oder vertikal zentriert (dies ist die Standard-Einstellung).

Die Defaultgröße des Dialogs lässt sich über die Variablen [EditBox\_Width] (Breite) und [EditBox\_Height] (Höhe) setzen.

Die Angabe bei Positionierungs-/Dimensionswerten erfolgt in Pixel.

Das Dialog-Icon kann global für alle Dialoge über die Variable [app\_icon] auf eine externe .ICO-Datei gesetzt werden.

## LISTBOX

### Syntax:

LISTBOX Eintrag = Titel, Elementenliste

### Beschreibung:

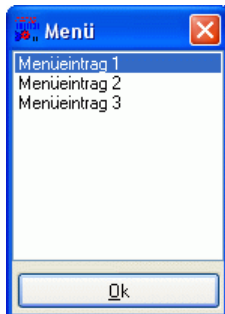
Zeigt einen Listenauswahldialog an, der dem Benutzer das Wählen von einem oder mehreren Einträgen aus einer vorgegebenen Liste an Elementen ermöglicht.

Als Parameter erwartet diese Funktion einen Dialogtitel sowie eine Liste an Elementen, die zur Auswahl im Listefeld angezeigt werden. Jedes Element wird dabei durch das in der Variablen [std\_sep] definierte Separationszeichen getrennt, dies ist normalerweise das Pipe-Zeichen (|).

Rückgabewert der Funktion ist der gewählte Listeneintrag.

Klickt der Benutzer den Schließen-Button des Dialogfensters oder drückt er auf "OK", ohne etwas auszuwählen, wird ein Leerstring zurückgegeben.

Standardmäßig erlaubt LISTBOX nur die Auswahl eines einzelnen Eintrags. Wenn jedoch die Variable [ListBox\_MultiSel] den Wert 0 hat, so lassen sich auch mehrere Einträge selektieren. In diesem Fall gibt die Funktion eine Liste zurück, in der alle selektierten Elemente durch das in [std\_sep] definierte Zeichen voneinander getrennt sind.



## Beispiele:

```
rem LISTBOX mit einfacher Auswahl
listbox [menu] = 'Menü', 'Pizza|Pommes|Frites|Döner|Hamburger|Hot Dog'
if [menu] != ''
    echo 'Sie haben ' # [menu] # ' bestellt!'
```

```

else
    echo 'Sie haben keine Auswahl getroffen.'
endif

rem LISTBOX mit Mehrfach-Auswahl
[ListBox_MultiSel] = [true]
listbox [menu] = 'Menü 2', 'Pizza|Pommes Frites|Döner|Hamburger|Hot Dog'
if [menu] != ''
    replacevar [menu] = [menu], [std_sep], [new_line]
    echo 'Ihre Bestellung: ' # [new_line] # [new_line] # [menu]
else
    echo 'Sie haben keine Auswahl getroffen.'
endif

```

## Bemerkungen:

Die Beschriftung des OK-Buttons von LISTBOX kann über die Variable [ListBox\_Ok] geändert werden. Wird die Beschriftung des Buttons auf einen Leerstring gesetzt, wird der Button aus dem Dialog ausgeblendet (nur Anzeige der Elemente in der Liste).

Die Position des Dialogs kann mit Hilfe der Variablen [ListBox\_X] und [ListBox\_Y] verändert werden. Hat [ListBox\_X] und/oder [ListBox\_Y] den Wert 0 wird der Dialog automatisch jeweils horizontal oder vertikal zentriert (dies ist die Standard-Einstellung).

Die Defaultgröße des Dialogs lässt sich über die Variablen [ListBox\_Width] (Breite) und [ListBox\_Height] (Höhe) setzen.

Die Angabe bei Positionierungs-/Dimensionswerten erfolgt in Pixel.

Das Dialog-Icon kann global für alle Dialoge über die Variable [app\_icon] auf eine externe .ICO-Datei gesetzt werden.

## FOLDERBOX

### Syntax:

FOLDERBOX Verzeichnis = Aufforderungstext, Vorgabepfad

### Beschreibung:

FOLDERBOX zeigt einen Standard-Auswahldialog zum Auswählen eines Verzeichnisses an.

Als Parameter erwartet diese Funktion einen Aufforderungstext sowie einen optionalen Vorgabepfad. Rückgabewert ist das gewählte Verzeichnis oder ein Leerstring, wenn kein Ordner gewählt wurde oder der Benutzer den "Abbrechen"-Button klickt.



## Beispiele:

```
folderbox [pfad] = 'Bitte wählen Sie ein Verzeichnis!', 'C:\jmksf\rb5'

if [pfad] ! ''
    echo 'Das gewählte Verzeichnis lautet: ' # [pfad]
else
    echo 'Kein Verzeichnis gewählt.'
endif
```

## Bemerkungen:

Verwenden Sie die Dialoge OPENBOX und SAVEBOX, um die Auswahl einer Datei zu ermöglichen.

## OPENBOX, SAVEBOX

### Syntax:

OPENBOX Datei = Dialogtitel, Filter  
SAVEBOX Datei = Dialogtitel, Filter

### Beschreibung:

OPENBOX zeigt einen Standarddialog zum Öffnen einer Datei an.

SAVEBOX zeigt einen Standarddialog zum Speichern einer Datei an.

Als Parameter benötigen beide Funktionen einen Dialogtitel sowie einen Dateifilter, der wie folgt definiert wird:

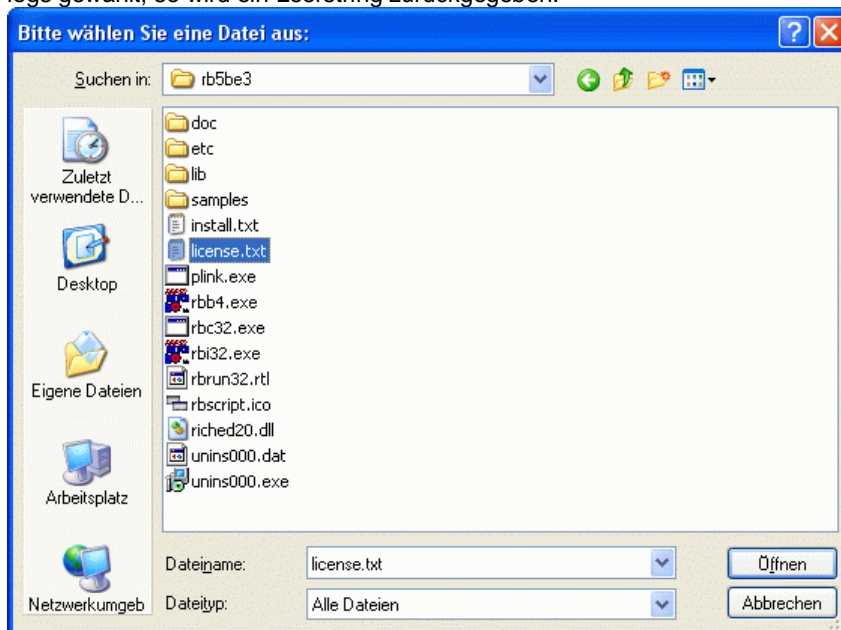
Filterbezeichnung|Filter

Beispiel für einen TXT-Dateifilter wäre: "Textdateien|\*.txt".

Um mehrere Filter zur Verfügung zu stellen (diese können im Dialog vom Benutzer ausgewählt werden) lassen sich die einzelnen Filterelemente einfach hintereinander schreiben: "Textdateien|\*.txt|Word-Dateien|\*.doc" ...

Um mehrere Filter unter einem Namen zu verwalten, spezifizieren Sie den Filter wie folgt: "Ausführbare Dateien|\*.exe;\*.com;\*.bat".

Rückgabewert der Funktion ist der komplette Dateiname mit Pfad, wenn der Benutzer eine Datei doppelt geklickt oder einen Dateinamen ausgewählt und OK gedrückt hat. Hat der Benutzer Abbrechen oder den Schließen-Button des Dialogs gewählt, so wird ein Leerstring zurückgegeben.



## Beispiele:

```
openbox [dateiname] = 'Bitte Datei wählen!', 'Alle Dateien (*.*)|*.*'

if [dateiname] ! ''
    echo 'Der gewählte Dateiname lautet: ' # [dateiname]
else
    echo 'Keine Datei ausgewählt.'
endif
```

## Bemerkungen:

Wenn Sie die vordeklarierte Variable [OpenBox\_MultiSel] (bei OPENBOX) bzw. [SaveBox\_MultiSel] (bei SAVEBOX) auf den Wert 0 setzen, können in dem jeweiligen Dialog mehrere Dateien ausgewählt werden. Ist dies der Fall, ist der Rückgabewert eine Liste, beginnend mit dem Pfadnamen und den einzelnen Dateien, jeweils durch Pipe-Zeichen (|) getrennt.

Das Pipe-Zeichen (|) ist der Standard-Separator zum Trennen von Listenelementen in einem String. Dieser kann über die Variable [std\_sep] geändert werden, was dann für alle Listen, die von RapidBATCH verarbeitet werden, gilt.

## INFOBOX

### Syntax:

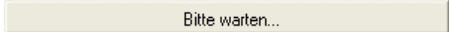
INFOBOX Meldung, Anzeigemodus

### Beschreibung:

INFOBOX zeigt eine Meldung auf dem Bildschirm an.

Dieser Dialog stellt ein titellostes Fenster bereit, welches als Status-Anzeige oder ähnliches benutzt werden kann. Beispielsweise kann solch ein Fenster mit dem Text "Bitte warten..." eingeblendet werden, bis verschiedene Operationen im Hintergrund durchgeführt wurden. Der Script-Programmierer ist daher für die rechtzeitige Ein- bzw. Ausblendung der Meldung selbst verantwortlich. Der INFOBOX-Dialog wird als Vordergrundfenster vor allen anderen Anwendungen auf dem Bildschirm angezeigt und lässt sich vom Benutzer weder verschieben noch in den Hintergrund bringen.

Als Parameter erwartet diese Anweisung den gewünschten Informationstext sowie den Modus, ob die Information eingeblendet oder ausgeblendet werden soll, wobei 0 oder die build-in Konstante "show" für einblenden (Dialog anzeigen) und -1 (oder "hide") für ausblenden (Dialog verstecken) steht. Selbiges kann auch durch Setzen der Variablen [InfoBox\_Enabled] auf den gewünschten Status erzielt werden.



## Beispiele:

```
rem INFOBOX einblenden
infobox 'Das Script wird in 5 Sekunden fortgesetzt...', 'show'

rem 5 Sekunden warten
wait '5000'

rem INFOBOX wieder ausblenden
[InfoBox_Enabled] = 'hide'
```

## Bemerkungen:

Die Position des Dialogs kann mit Hilfe der Variablen [InfoBox\_X] und [InfoBox\_Y] verändert werden. Hat [InfoBox\_X] und/oder [InfoBox\_Y] den Wert 0, so wird der Dialog automatisch jeweils horizontal oder vertikal zentriert (dies ist die Standard-Einstellung).

Die Defaultgröße des Dialogs lässt sich über die Variablen [InfoBox\_Width] (Breite) und [InfoBox\_Height] (Höhe) setzen.

Die Angabe bei Positionierungs-/Dimensionswerten erfolgt in Pixel.

## TRAYMENU

### Syntax:

TRAYMENU Auswahl = Hilfstext, Elementenliste

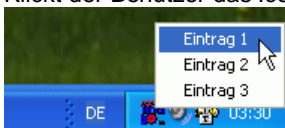
### Beschreibung:

Zeigt ein Programm-Menü als Icon in der Windows-Taskleiste an, aus dem der Benutzer via Rechtsklick ein Menü anzeigen lassen und einen Menüpunkt auswählen kann.

Als Parameter erwartet diese Funktion einen Hilfstext, der als Tooltip angezeigt wird, sobald die Maus über das Tray-Icon bewegt wird, sowie eine Liste an Elementen, die zur Auswahl im Menü angezeigt werden. Jedes Element wird dabei durch das in der Variablen [std\_sep] definierte Separationszeichen getrennt, dies ist normalerweise das Pipe-Zeichen (|).

Rückgabewert der Funktion ist der gewählte Menüpunkt.

Klickt der Benutzer das Icon mit der linken Maustaste an, wird ein Leerstring zurückgegeben.



### Beispiele:

```
traymenu [menu] = 'Was möchten Sie essen?', 'Pizza|Pommes Frites|Döner|Hamburger|Hot Dog'

if [menu] ! ''
    echo 'Ihre Auswahl lautet: ' # [menu]
else
    echo 'Sie haben keine Auswahl getroffen.'
endif
```

### Bemerkungen:

In dem Menü, was beim Rechtsklick angezeigt wird, können auch Shortcuts definiert werden, die durch Tastendruck angesteuert werden können. Um einen Shortcut-Key zu definieren, schreiben Sie einfach ein &-Zeichen vor das jeweilige Element, z.B. "B&enden". Hier wird das "e" unterstrichen dargestellt und kann mit ALT+e ausgewählt werden. Im Menü kann auch eine Trennlinie angegeben werden, geben Sie dazu einfach einen Bindestrich (-) als Element in der Liste an.

Das Icon kann global für alle Dialoge über die Variable [app\_icon] auf eine externe .ICO-Datei gesetzt werden.

## Systemspezifische Operationen

### CHDIR, MKDIR, RMDIR

**Syntax:**

```
CHDIR Pfad  
MKDIR Pfad  
RMDIR Pfad
```

**Beschreibung:**

CHDIR wechselt das aktuelle Arbeitsverzeichnis in das gegebene Verzeichnis, MKDIR legt das gegebene Verzeichnis neu an.

Mit RMDIR wird das gegebene Verzeichnis entfernt, wobei dieses komplett leer sein muss und keine Unterverzeichnisse enthalten darf.

**Beispiele:**

```
chdir 'c:\jmksf\rb5'  
mkdir 'c:\test'  
rmdir 'c:\test'
```

**Bemerkungen:**

War die jeweilige Operation erfolgreich, so erhält die Variable [errorcode] den Wert 0, andernfalls -1. Mit dieser Möglichkeit kann z.B. geprüft werden, ob ein Verzeichnis existiert oder nicht.

Um gesamte Verzeichnisse mitsamt Unterverzeichnissen und Dateien zu löschen, verwenden Sie die RapidBATCH Library-Funktion DELDIR, die in der SYSTEM.RB Bibliothek definiert ist.

### SYSTEM

**Syntax:**

```
SYSTEM Kommando
```

**Beschreibung:**

Führt ein Betriebssystemkommando oder Programm aus. Dies sind unter Windows reine DOS-Befehle. Das Script wartet solange mit der Ausführung bis das mit SYSTEM gestartete Programm beendet wurde.

**Beispiele:**

```
system 'dir >list.txt'
```

**Bemerkungen:**

Die Variable [errorcode] erhält den Wert -1, wenn kein COMMAND.COM oder CMD.EXE gefunden wurde, andernfalls 0.

### SHELL

**Syntax:**

```
SHELL Programmpfad, Anzeigemodus
```

**Beschreibung:**

Führt ein externes Programm aus.

Das Script wird dabei fortgeführt, so dass das gestartete Programm mit dem Script parallel läuft (Multitasking). Die Anweisung erwartet einen Programmpfad (.EXE, .COM oder .BAT) und einen Anzeigemodus, der entweder durch eine Zahl oder eine der build-in Konstanten repräsentiert wird. Mögliche Werte sind SHOW (0), HIDE (1), MAXIMIZED (2) und MINIMIZED (3).

**Beispiele:**

```
shell 'c:\windows\calc.exe', 'show'  
shell 'c:\windows\desktop\test.bat', 'hide'
```

**Bemerkungen:**

[errorcode] erhält einen Wert > 32 (die Prozess-ID des Programms), wenn das Programm erfolgreich gestartet werden konnte.

## CALL

**Syntax:**

CALL Programmpfad, Anzeigemodus

**Beschreibung:**

Startet ein Programm im Modalmodus. Das bedeutet, dass das RapidBATCH-Script mit der Ausführung solange wartet, bis das mit CALL gestartete Programm beendet wurde.

Äquivalent zu SHELL benötigt CALL einen Anzeigemodus-Parameter, der entweder durch eine Zahl oder eine der build-in Konstanten repräsentiert wird. Mögliche Werte sind SHOW (0), HIDE (1), MAXIMIZED (2) und MINIMIZED (3), um Programme beispielsweise minimiert oder versteckt zu starten.

**Beispiele:**

```
echo 'Starten von Notepad...'  
call 'notepad.exe', 'show'  
echo 'Notepad wurde beendet.'  
end
```

**Bemerkungen:**

[errorcode] erhält -1, wenn ein Fehler auftrat, ansonsten den Rückgabewert des gestarteten Programms.

## OPEN

**Syntax:**

OPEN Pfad

**Beschreibung:**

OPEN öffnet eine Datei mit dem dafür vorgesehenen Programm oder eine Internetseite im Browser. Mit OPEN können auch externe Programme gestartet werden.

**Beispiele:**

```
open 'hello.doc'  
open 'http://www.jmksf.com/'  
open 'c:\windows\calc.exe'  
open 'c\jmksf\rb5\samples\hello.rb'
```

**Bemerkungen:**

keine

## NEWFILE, DELFILE

**Syntax:**

NEWFILE Dateiname  
DELFIL Dateiname



**Beschreibung:**

NEWFILE legt eine neue, leere Datei an. Existiert die Datei bereits, wird Sie mit der neuen, leeren Datei überschrieben.  
DELFILE löscht eine vorhandene Datei unwiderruflich.

**Beispiele:**

```
rem Neue Datei anlegen
newfile 'neu.txt'

rem Temp-Datei löschen
delfile 'ausgabe.tmp'
```

**Bemerkungen:**

Ist die jeweilige Operation erfolgreich, so erhält [errorcode] den Wert 0, ansonsten -1.

## COPYFILE, RENAMEFILE

**Syntax:**

COPYFILE Quelldatei, Zieldatei  
RENAMEFILE Quelldatei, Zieldatei

**Beschreibung:**

COPYFILE kopiert eine Quelldatei in eine Zieldatei.  
RENAMEFILE verschiebt oder benennt eine Quelldatei in eine Zieldatei um.

**Beispiele:**

```
rem Datei kopieren...
copyfile 'c:\jmksf\rb5\samples\hello.rb', 'a:\hello.rb'

rem Datei umbenennen...
renamefile 'c:\test.neu', 'c:\test.old'
```

**Bemerkungen:**

Ist die jeweilige Operation erfolgreich, so erhält [errorcode] den Wert 0, ansonsten -1.

## FILESIZE

**Syntax:**

FILESIZE Dateigröße = Dateiname

**Beschreibung:**

Ermittelt die Größe einer Datei und gibt diese Größe in Bytes zurück.

**Beispiele:**

```
filesize [bytes] = 'c:\command.com'
echo 'COMMAND.COM hat eine Größe von ' # [bytes] # ' Bytes'
end
```

**Bemerkungen:**

Um eine Dateigröße in Kilo-, Mega-, Giga- oder Terabyte auszugeben muss der Wert jeweils durch 1024 geteilt werden.  
Für Kilobyte teilt man 1-mal durch 1024, für Terabyte 4-mal durch 1024, usw.

## FILEEXISTS

### Syntax:

FILEEXISTS Existenz = Dateiname

### Beschreibung:

Prüft die Existenz der angegebenen Datei. Wurde die Datei gefunden, gibt die Funktion den Wert 0, ansonsten -1 zurück.

### Beispiele:

```
fileexists [errorcode] = 'c:\windows\calc.exe'
if [errorcode] = '0' echo 'Die Datei CALC.EXE existiert!'
```

### Bemerkungen:

keine

## READFILE

### Syntax:

READFILE Zeile = Dateiname, Zeilennummer

### Beschreibung:

Liest die angegebene Zeile aus der gegebenen Datei und gibt diese als String zurück. Ist das Dateiende erreicht, erhält [errorcode] den Wert -1, und der Rückgabestring ist "EOF" (= "End Of File"). Wenn die Datei nicht gefunden wurde, erhält nur [errorcode] den Wert -1, der Rückgabewert ist dann ein Leerstring.

Wenn als Zeilennummer ein Wert kleiner 1 angegeben wird, liest READFILE den Inhalt der gesamten Datei und gibt diesen zurück.

### Beispiele:

```
rem Erste Zeile lesen...
readfile [text] = 'c:\test.txt', '1'

rem Gesamte Datei lesen...
readfile [text] = 'c:\test.txt', '0'

rem Ersten 10 Zeilen einer Datei ausgeben
[i] = '0'
[text] = ''
repeat
    [i] + '1'
    readfile [zeile] = 'C:\jmksf\rb5\readme.txt',[i]
    if [zeile] != 'EOF'
        [text] # [zeile] # [new_line]
    endif
until [i] = '10' | [zeile] = 'EOF'

echo [text]
end
```

### Bemerkungen:

Verwenden Sie die GETFILE-Funktion, um Daten auf Byte-Ebene aus einer Datei (z.B. Binärdatei) zu lesen.

## WRITEFILE

### Syntax:

WRITEFILE Dateiname, Stringausdruck

**Beschreibung:**

Schreibt den angegebenen Stringausdruck in die ebenfalls angegebene Datei, wobei die Daten ans Dateiende angehängt werden. Der Inhalt der Datei wird dabei nicht überschrieben, sondern nur erweitert. Wenn die Datei nicht existiert, wird sie automatisch erstellt.

**Beispiele:**

```
writefile 'c:\test.txt', 'Hello World'
```

**Bemerkungen:**

Die Variable [errorcode] erhält den Wert -1, wenn die Datei nicht erstellt oder geschrieben werden konnte, andernfalls 0.

Standardmäßig hängt WRITEFILE einen Windows-Zeilenumbruch (carriage-return und line-feed) an den zu schreiben den String an. Dies kann über die Variable [eol] (end-of-line) dynamisch angepasst werden, so dass beliebige Zeichen als Zeilenende (z.B. für Unix- oder Mac-Dateien) (oder auch gar kein Zeichen) angehängt werden.

Verwenden Sie die PUTFILE-Anweisung, um Daten auf Byte-Ebene in eine Datei (z.B. Binärdatei) zu schreiben.

## GETFILE

**Syntax:**

GETFILE Liste-von-Zeichencodes = Dateiname, Start-Position, Byte-Anzahl

**Beschreibung:**

Liest Daten auf Byte-Ebene aus einer Datei.

Neben dem Dateinamen benötigt diese Funktion ein Anfangsbyte, welches als Start-Position übergeben wird, und eine Anzahl an Bytes, wieviele Zeichen von Start-Position aus gelesen werden sollen.

Ein ASCII-Zeichen entspricht in einer Datei einem Byte und kann maximal einen Wert von 255 annehmen (dies entspricht einem Binärwert von "1111 1111", also 8 Bit = 1 Byte).

Rückgabewert ist eine Liste an ASCII-Zeichencodes, die jeweils durch das RapidBATCH-Listentrennzeichen voneinander getrennt sind (im Normalfall das Pipe-Zeichen (|)).

**Beispiele:**

```
rem Beispielscript zur Demonstration von Datei Lese- und Schreiboperationen auf Byteebene
newfile 'test.txt'
rem schreiben von "Hello World" (Liste von ASCII-Zeichencodes)
putfile 'test.txt', '1', '72|101|108|108|111|32|87|111|114|108|100'

rem Lesen des Wortes "World" als Liste an ASCII-Zeichencodes
getfile [wort] = 'test.txt', '7', '5'
echo 'Gelesene Zeichencodes: ' # [wort]

rem Konvertierung und Zusammenfügung der ASCII-Werte zu Klartext
[i] = '0'
repeat
    [i] + '1'
    gettok [ch] = [wort], '|', [i]
    if [ch] != ''
        getchr [ch] = [ch]
        [klartext] # [ch]
    endif
until [ch] = ''

rem Anzeigen des Klartextes:
echo 'Klartext: ' # [klartext]
```

**Bemerkungen:**

Verwenden Sie die PUTFILE-Anweisung, um einen Wert auf Byte-Ebene in eine Datei zu schreiben.

Schlägt GETFILE fehl, erhält [errorcode] den Wert -1 und ein Leerstring wird zurückgegeben. Andernfalls erhält [errorcode] den Wert 0.

## PUTFILE

### Syntax:

PUTFILE Dateiname, Start-Position, Liste-von-Zeichencodes

### Beschreibung:

Schreibt einen Stringausdruck auf Byte-Ebene in eine Datei.

Als Parameter wird ein Dateiname, eine Byte-Startposition in der Datei (wo der Schreibvorgang begonnen werden soll) und eine Liste an ASCII-Zeichencodes, die ab Start-Position als Zeichen byteweise in die Datei geschrieben werden sollen, erwartet.

Ein ASCII-Zeichen entspricht in einer Datei einem Byte und kann maximal einen Wert von 255 annehmen (dies entspricht einem Binärwert von "1111 1111", also 8 Bit = 1 Byte).

### Beispiele:

```
rem Schreiben einer Datei auf Byte-Ebene

rem Nicht existente Datei MUSS zuerst explizit erzeugt werden!
newfile 'bytefile.bin'

[klartext] = 'Dies ist der Klartext, der in die Datei geschrieben wird!'

rem Konvertierung des Klartextes in eine Liste an ASCII-Zeichencodes
[i] = '0'
getlen [len] = [klartext]
repeat
    [i] + '1'
    getcharat [ch] = [klartext], [i]
    getasc [ch] = [ch]
    if [byteliste] ! '' [byteliste] # '|'
        [byteliste] # [ch]
until [i] = [len]

echo 'Klartext: ' # [klartext]
echo 'Liste an ASCII-Zeichencodes: ' # [byteliste]

rem Schreiben der Datei
putfile 'bytefile.bin', '1', [byteliste]

if [errorcode] = [false]
    echo 'PUTFILE ist fehlgeschlagen!'
endif
```

### Bemerkungen:

Die zu schreibende Datei muss existieren. Wenn die Datei nicht existiert, kann auch kein Wert in sie hineingeschrieben werden. Erstellen Sie dann einfach eine neue Datei mit Hilfe von NEWFILE.

[errorcode] erhält den Wert -1, wenn PUTFILE fehlschlug, andernfalls 0.

## FINDFILES

### Syntax:

FINDFILES Dateiliste = Dateifilter, Verzeichnistiefe

### Beschreibung:

Liest alle Dateien eines Verzeichnisses (mitsamt Unterverzeichnissen) aus und gibt diese als Dateiliste zurück.

Diese Funktion benötigt als Parameter einen so genannten Dateifilter (der die Wildcards \* und ? beinhalten kann) und eine Ebenen-Angabe (Verzeichnistiefe), die angibt, wieviele Unterverzeichnis-Ebenen maximal mit eingelesen werden sollen. Wird als Verzeichnistiefe der Wert 0 übergeben, liest FINDFILES alle Verzeichnisse bis zu 255 Unterverzeichnis-Ebenen ein. Wird der Wert 1 als Tiefe übergeben, so wird nur das aktuelle Verzeichnis eingelesen, bei dem Wert 2 wird das aktuelle Verzeichnis mit einer Unterverzeichnis-Ebene eingelesen, usw..

Rückgabewert ist eine Liste an Dateien (mit relativen Unterverzeichnis-Pfaden), die durch den Listentrenner (definiert in der Variable [std\_sep]) voneinander getrennt sind.

**Beispiele:**

```
rem Alle Dateien im Verzeichnis RB5 (ohne Unterverzeichnisse) auflisten
chdir 'c:\jmksf\rb5'
findfiles [filelist] = '*.*', '1'
replacevar [filelist] = [filelist], [std_sep], [new_line]
echo [filelist]

rem Alle EXE-Dateien von C:\JMKSF auflisten
findfiles [filelist] = 'c:\jmksf\*.exe', '0'
replacevar [filelist] = [filelist], [std_sep], [new_line]
echo [filelist]
end
```

**Bemerkungen:**

Wenn Sie die Dateien zur späteren Verwendung aus der Dateiliste extrahieren wollen, benutzen Sie bitte die Funktionen GETTOK und REPLACEVAR, wie in dem Beispiel gegeben.

Die Funktionsbibliothek ARRAY.RB bietet ausserdem nützliche Funktionen und Prozeduren zum Umwandeln von Dateilisten in Arrays.

## GETFILEATT, LETFILEATT

**Syntax:**

GETFILEATT Attributschalter = Dateiname, Attributnummer  
LETFILEATT Dateiname, Attributnummer

**Beschreibung:**

GETFILEATT gibt Auskunft, ob ein angefordertes Dateiaattribut bei der ebenfalls spezifizierten Datei gesetzt ist oder nicht. Mögliche Werte für die Dateiaattributnummer sind NORMAL (0), ARCHIVED (1), READONLY (2), COMPRESSED (3), DIRECTORY (4), HIDDEN (5) und SYSTEM (6) (zu Deutsch: Normal, Archiviert, Schreibgeschützt, Komprimiert, Verzeichnis, Versteckt und Systemdatei). Neben den numerischen Werten lassen sich optional auch die englischen Bezeichner als Attribut-Typ angeben, z.B. ARCHIVED oder READONLY.

GETFILEATT gibt 0 zurück, wenn das abgefragte Dateiaattribut gesetzt ist, andernfalls -1.

LETFILEATT setzt Dateiaattribute, wenn diese nicht gesetzt sind. Mögliche Werte für die Dateiaattribute sind identisch zu denen von GETFILEATT, allerdings lassen sich die Attribute COMPRESSED (3) und DIRECTORY (4) nicht setzen. Schlägt LETFILEATT fehl, so erhält [errorcode] den Wert -1, ansonsten 0.

**Beispiele:**

```
[file] = '..\readme.txt'

rem Attribute auslesen
[i] = '1'
repeat
  getfileatt [ret] = [file],[i]
  [res] = [res] # [i] # ' = ' # [ret] # [new_line]
  [i] + '1'
until [i] > '6'

echo 'File Attributes of ' # [file] # [new_line] # [new_line] # [res]

rem Attribute setzen
letfileatt [file], 'normal'

end
```

**Bemerkungen:**

keine

## GETFILEDATE, LETFILEDATE

### Syntax:

GETFILEDATE Datum = Dateiname  
LETFILEDATE Dateiname, Datum

### Beschreibung:

GETFILEDATE liest das Änderungsdatum einer Datei und gibt dieses im Format "mmddyyyy" zurück.

LETFILEDATE setzt das Änderungsdatum einer Datei. Parameter sind der Dateiname und ein Datum im Format "mmddyyyy".

### Beispiele:

```
newfile 'test.txt'
getfiledate [d] = 'test.txt'

echo 'test.txt wurde geändert am ' # [d]

rem Nun setzen wir das Datum auf den 01.02.2004
letfiledate 'test.txt', '02012004'

getfiledate [d] = 'test.txt'
echo 'test.txt wurde "geändert" am ' # [d]
delfile 'test.txt'
end
```

### Bemerkungen:

Wurde die Datei nicht gefunden, erhält [errorcode] sowohl von LETFILEDATE als auch von GETFILEDATE den Wert -1, ansonsten 0. Der Rückgabewert von GETFILEDATE ist in solch einem Fall ein Leerstring.

Verwenden Sie GETFILETIME und LETFILETIME um die Änderungsuhrzeit einer Datei zu lesen oder zu setzen.

## GETFILETIME, LETFILETIME

### Syntax:

GETFILETIME Uhrzeit = Dateiname  
LETFILETIME Dateiname, Uhrzeit

### Beschreibung:

GETFILETIME liest die Änderungszeit einer Datei und gibt diese im Format "hhmmss" zurück.

LETFILETIME setzt die Änderungszeit einer Datei. Parameter sind der Dateiname und eine Zeit im Format "hhmmss".

### Beispiele:

```
newfile 'test.txt'
getfiletime [t] = 'test.txt'

echo 'test.txt wurde geändert um ' # [t]

rem Nun setzen wir die Uhrzeit auf 14:45:00
letfiletime 'test.txt', '144500'

getfiletime [t] = 'test.txt'
echo 'test.txt wurde "geändert" um ' # [t]
delfile 'test.txt'
end
```

### Bemerkungen:

Wurde die Datei nicht gefunden, erhält [errorcode] sowohl von LETFILETIME als auch von GETFILETIME den Wert -1, ansonsten 0. Der Rückgabewert von GETFILETIME ist in solch einem Fall ein Leerstring.

Verwenden Sie GETFILEDATE und LETFILEDATE, um ein Dateidatum zu lesen oder zu setzen.

## SHORTCUT

### Syntax:

SHORTCUT Verknüpfungsdatei, Verknüpfungspfad, Parameter, Arbeitsverzeichnis, Icon, Anzeigemodus

### Beschreibung:

Erstellt eine Verknüpfung.

Die Verknüpfung wird unter dem als Verknüpfungsdatei angegebenen Pfad gespeichert, wobei die Dateiendung .LNK automatisch angehängt wird. Programmpfad gibt den Pfad zu einer Datei, auf die die Verknüpfung zeigen soll, an. Parameter definiert optionale Kommandozeilenparameter, Arbeitsverzeichnis ein optionales Arbeitsverzeichnis, Icon einen optionalen Icon-Pfad. Der Anzeigemodus kann 'SHOW', 'MINIMIZED' oder 'MAXIMIZED' sein. 'HIDE' ist bei Verknüpfungen als Anzeigemodus-Wert nicht möglich.

### Beispiele:

```
[userdir] = 'C:\Dokumente und Einstellungen\' # [curuser] # '\Desktop'
[prog] = 'C:\jmkfs\rb5\rb4.exe'

shortcut [userdir] # '\RapidBATCH Schortcut', [prog], '', '', '', 'show'

if [errorcode] = [true]
    echo 'Verknüpfung wurde erfolgreich erstellt!'
else
    echo 'Fehler beim Erstellen der Verknüpfung.'
endif
```

### Bemerkungen:

[errorcode] erhält den Wert 0, wenn kein Fehler auftrat, ansonsten -1.

## DISKFREE, DISKSIZE

### Syntax:

DISKFREE Speicherplatz = Laufwerksbuchstabe

DISKSIZE Speicherplatz = Laufwerksbuchstabe

### Beschreibung:

DISKFREE gibt den freien verfügbaren Speicher eines Datenträgers in Bytes zurück.

DISKSIZE gibt die gesamte Speicherkapazität eines Datenträgers in Bytes zurück.

### Beispiele:

```
diskfree [df] = 'C:\'
echo 'Es sind noch ' # [df] # ' Bytes freier Festplattenspeicher auf Laufwerk C: verfü-
bar.'
diskfree [ds] = 'C:\'
echo 'Laufwerk C: hat ein gesamtes Volumen von ' # [ds] # ' Bytes.'
end
```

### Bemerkungen:

Um den freien Speicher in Kilo-, Mega-, Giga- oder Terabyte auszugeben, muss der Wert jeweils durch 1024 geteilt werden. Für Kilobyte teilt man 1-mal durch 1024, für Terabyte 4-mal durch 1024, usw.

## DISKTYPE

### Syntax:

DISKTYPE Laufwerkstypen-ID = Laufwerksbuchstabe



**Beschreibung:**

Gibt Auskunft über den Typ eines Speichermediums. Der Rückgabewert ist einer der folgenden Werte:

- 0 = Unbekanntes Medium
- 1 = Wechselspeicher
- 2 = Fixiertes Laufwerk (z.B. Festplatte)
- 3 = Remote Laufwerk
- 4 = CD-ROM
- 5 = RAM-Disk (z.B. USB-Stick)

**Beispiele:**

```
disktype [type] = 'C:\'  
if [type] = '0' [type] = 'unbekanntes '  
if [type] = '1' [type] = 'Wechselspeicher-'  
if [type] = '2' [type] = 'fixiertes '  
if [type] = '3' [type] = 'Remote-'  
if [type] = '4' [type] = 'CD-ROM-'  
if [type] = '5' [type] = 'RAM-Disk-'  
  
echo 'Laufwerk C: ist ein ' # [type] # 'Medium'  
end
```

**Bemerkungen:**

keine

## GETENV, LETENV

**Syntax:**

GETENV Wert = Umgebungsvariable  
LETENV Umgebungsvariablenzuweisung

**Beschreibung:**

GETENV ermöglicht das Auslesen von Umgebungsvariablen. Die Funktion erwartet eine Variable, die den Wert der auszulesende Umgebungsvariable erhält sowie den Namen der Umgebungsvariable, die gelesen werden soll. LETENV erlaubt das Setzen/Erzeugen von Umgebungsvariablen. Dabei gibt man die Umgebungsvariable, gefolgt von einem Gleichheitszeichen oder Leerzeichen und einem Wert an (siehe Beispiel).

**Beispiele:**

```
rem PATH auslesen  
getenv [pathvar] = 'PATH'  
  
rem Neue Umgebungsvariable TEST erzeugen  
letenv 'TEST=Hello World'
```

**Bemerkungen:**

Geänderte oder neu erstellte Variablen werden nach Beendigung des Scripts durch das Betriebssystem automatisch wieder zurückgesetzt bzw. neue Variablen werden gelöscht.

## MCI

**Syntax:**

MCI Befehlsausdruck

**Beschreibung:**

Sendet ein Kommando an das Windows Media Control Interface, mit dem sich Mediadaten aller Art abspielen lassen, z.B. MIDI, AVI, WAV, MP3, MPEG, usw.

**Beispiele:**

```
mci 'open "music.mp3" '  
mci 'play "music.mp3" wait '  
mci 'close "music.mp3" '  
end
```

**Bemerkungen:**

Weitere Informationen zu MCI-Kommandos finden Sie in der englischsprachigen Hilfedatei MCI.HLP, die sich im \DOC-Verzeichnis Ihrer RapidBATCH-Installation befindet.

Die Variable [errorcode] erhält das Ergebnis der API-Funktion mciSendString(), und kann somit auf evtl. Fehler abgefragt werden.

## NEWREGKEY, DELREGKEY

**Syntax:**

NEWREGKEY Hauptschlüssel, Schlüsselpfad  
DELREGKEY Hauptschlüssel, Schlüsselpfad

**Beschreibung:**

NEWREGKEY legt einen neuen Schlüssel in der Registry an.

DELREGKEY löscht einen Schlüssel aus der Registry, mitsamt Unterschlüsseln.

Beide Anweisungen erwarten als Parameter einen Hauptschlüssel, der als einer der folgenden Werte übergeben werden muss:

- 'CLASSES\_ROOT' für HKEY\_CLASSES\_ROOT
- 'CURRENT\_CONFIG' für HKEY\_CURRENT\_CONFIG
- 'CURRENT\_USER' für HKEY\_CURRENT\_USER
- 'LOCAL\_MACHINE' für HKEY\_LOCAL\_MACHINE
- 'USERS' für HKEY\_USERS

'DYN\_DATA' für HKEY\_DYN\_DATA

**Beispiele:**

```
newregkey 'CURRENT_USER', 'Software\Test'  
if [errorcode] = '0'  
    echo 'Erstellen von HKEY_CURRENT_USER\Software\Test war erfolgreich.'  
endif  
  
delregkey 'CURRENT_USER', 'Software\Test'  
if [errorcode] = '0'  
    echo 'Löschen von HKEY_CURRENT_USER\Software\Test war erfolgreich.'  
endif  
end
```

**Bemerkungen:**

Verwenden Sie GETREGVAL und LETREGVAL, um Werte aus der Registry zu lesen oder Werte in die Registry zu schreiben.

## GETREGVAL

**Syntax:**

GETREGVAL Wert = Datentyp, Hauptschlüssel, Unterschlüssel, Wertbezeichner

**Beschreibung:**

GETREGVAL liest einen Wert aus der Registry. Parameterwerte sind der Datentyp des zu lesenden Registry-Wertes, welcher durch die Konstanten "STRING" (Standard), "DWORD" und "BINARY" definiert wird (auch repräsentiert durch die numerischen Werte 0, 1 und 2), der Hauptschlüssel, ein Unterschlüssel und der Bezeichner des Registry-Wertes, dessen Wert ausgelesen werden soll.

Als Hauptschlüssel sind folgende Werte als build-in Konstanten definiert:

- 'CLASSES\_ROOT' für HKEY\_CLASSES\_ROOT
- 'CURRENT\_CONFIG' für HKEY\_CURRENT\_CONFIG
- 'CURRENT\_USER' für HKEY\_CURRENT\_USER
- 'LOCAL\_MACHINE' für HKEY\_LOCAL\_MACHINE
- 'USERS' für HKEY\_USERS
- 'DYN\_DATA' für HKEY\_DYN\_DATA

Der Unterschlüssel gibt den Pfad des Unterschlüssels, jeweils getrennt durch Backslash (\), an.

Als Wertbezeichner wird der in der Registry definierte Bezeichner angegeben.

Rückgabewert ist der gelesene Wert im entsprechenden Datentypsformat. Binary-Werte werden durch eine mit dem Standard-Separator separierten Liste an ASCII-Zeichencodes zurückgegeben, wie dies bei der Funktion GETFILE der Fall ist.

**Beispiele:**

```
rem Wert in Registry schreiben
letregval 'STRING', 'CURRENT_USER', 'Software\jmkfs\RapidBATCH', 'HelloWorld', 'Hello
World by RapidBATCH ;)'

if [errorcode] = '0'
    echo 'Wert wurde erfolgreich geschrieben!'
else
    echo 'Fehler beim Schreiben.'
    halt
endif

rem Wert wieder auslesen
getregval [wert] = 'STRING', 'CURRENT_USER', 'Software\jmkfs\RapidBATCH', 'HelloWorld'
if [errorcode] = '0'
    echo 'Gelesener Wert: ' # [wert]
else
    echo 'Fehler beim Lesen des Wertes.'
endif

rem Aufäumen...
delregval 'CURRENT_USER', 'Software\jmkfs\RapidBATCH', 'HelloWorld'
if [errorcode] = '-1' echo 'Fehler beim Löschen von Registry-Wert'
end
```

**Bemerkungen:**

Schlägt GETREGVAL fehl, weil z.B. der zu lesende Wert nicht existiert, so erhält [errorcode] den Wert '-1' und gibt einen Leerstring zurück. Ansonsten erhält [errorcode] den Wert '0'.

Verwenden Sie LETREGVAL und DELREGVAL, um Registry-Werte zu schreiben oder zu löschen.

**LETREGVAL****Syntax:**

LETREGVAL Datentyp, Hauptschlüssel, Unterschlüssel, Wertbezeichner, Wert

**Beschreibung:**

LETREGVAL schreibt einen Wert in die Registry.

Parameterwerte sind der Datentyp, der Hauptschlüssel, der Unterschlüssel, der Bezeichner des Registry-Wertes und der zu schreibende Wert.

Existiert der Wert noch nicht in der Registry, so wird er neu erstellt, andernfalls geändert.

Als Datentyp können die build-in Konstanten "STRING" (Standard), "DWORD" und "BINARY" angegeben werden (auch repräsentiert durch die numerischen Werte 0, 1 und 2). Wird als Datentyp BINARY verwendet, so muss der Parameter Wert wie bei der Anweisung PUTFILE als eine ASCII-Zeichencodelliste übergeben wird, in der jeder zu schreibende Wert durch das Standard-Separationszeichen voneinander getrennt ist.

Als Hauptschlüssel sind folgende Werte als build-in Konstanten definiert:

- 'CLASSES\_ROOT' für HKEY\_CLASSES\_ROOT
- 'CURRENT\_CONFIG' für HKEY\_CURRENT\_CONFIG
- 'CURRENT\_USER' für HKEY\_CURRENT\_USER
- 'LOCAL\_MACHINE' für HKEY\_LOCAL\_MACHINE
- 'USERS' für HKEY\_USERS
- 'DYN\_DATA' für HKEY\_DYN\_DATA

Der Unterschlüssel gibt den Pfad des Unterschlüssels, jeweils getrennt durch Backslash (\), an.

Als Wertbezeichner wird ein frei definierbarer Bezeichner für den Wert definiert, über den der Wert später wieder ausgelesen werden kann.

Der letzte Parameter beinhaltet dann den Wert, der unter der Bezeichnung des Wertbezeichners in die Registry geschrieben wird, im als Datentyp entsprechend angegebenem Format.

**Beispiele:**

```
letregval 'STRING', 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'HelloWorld', 'Hello
World by RapidBATCH ;)'

if [errorcode] = '0'
    echo 'Wert wurde erfolgreich geschrieben!'
else
    echo 'Fehler beim Schreiben.'
    halt
endif

rem Schreiben von Binär 85 / Hexadezimal 55
getchr [bin] = '85'
letregval 'BINARY', 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'bin85', [bin]

if [errorcode] = '0'
    echo 'Wert wurde erfolgreich geschrieben!'
else
    echo 'Fehler beim Schreiben.'
    halt
endif

rem Aufäumen...
delregval 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'HelloWorld'
if [errorcode] = '-1' echo 'Fehler beim Löschen von Registry-Wert'
delregval 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'bin85'
if [errorcode] = '-1' echo 'Fehler beim Löschen von Registry-Wert'

end
```

**Bemerkungen:**

Wenn LETREGVAL fehlschlug, wird an [errorcode] der Wert -1, ansonsten 0 übergeben.

Verwenden Sie GETREGVAL und DELREGVAL, um Registry-Werte zu lesen oder zu löschen.

## DELREGVAL

### Syntax:

DELREGVAL Hauptschlüssel, Unterschlüssel, Bezeichner

### Beschreibung:

Löscht einen Registry-Wert.

DELREGVAL erwartet als Parameter einen Hauptschlüssel, einen Unterschlüssel sowie den Bezeichner des Wertes in der Registry, der gelöscht werden soll.

Als Hauptschlüssel sind folgende Werte als build-in Konstanten definiert:

- 'CLASSES\_ROOT' für HKEY\_CLASSES\_ROOT
- 'CURRENT\_CONFIG' für HKEY\_CURRENT\_CONFIG
- 'CURRENT\_USER' für HKEY\_CURRENT\_USER
- 'LOCAL\_MACHINE' für HKEY\_LOCAL\_MACHINE
- 'USERS' für HKEY\_USERS
- 'DYN\_DATA' für HKEY\_DYN\_DATA

Der Unterschlüssel gibt den Pfad des Unterschlüssels, jeweils getrennt durch Backslash (\), an.

Als Wertbezeichner wird der in der Registry definierte Bezeichner angegeben, der gelöscht werden soll.

### Beispiele:

```
rem Testwert schreiben
letregval 'STRING', 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'HelloWorld', 'Hello
World by RapidBATCH ;)'

if [errorcode] = '0'
    echo 'Wert wurde erfolgreich geschrieben!'
else
    echo 'Fehler beim Schreiben.'
    halt
endif

rem Testwert wieder entfernen
delregval 'CURRENT_USER', 'Software\jmksf\RapidBATCH', 'HelloWorld'
if [errorcode] = '-1' echo 'Fehler beim Löschen von Registry-Wert'
end
```

### Bemerkungen:

Wenn das Löschen des Wertes fehlschlug, erhält [errorcode] den Wert -1, ansonsten 0.

Verwenden Sie LETREGVAL und GETREGVAL, um Registry-Werte zu lesen bzw. zu schreiben.

## SENDKEYS

### Syntax:

SENDKEYS Fenstertitel, Fernsteuerungsbefehle

### Beschreibung:

Sendet Tastaturbefehle an das Fenster mit dem jeweiligen Fenstertitel. Diese Anweisung ermöglicht somit das Fernsteuern beliebiger Windows-Programme mit Hilfe einer einfach definierten Makrosprache.

Im Grunde ist diese Fernsteuerungs-Makrosprache (SENDKEYS-Script) sehr einfach aufgebaut: Man muss die Tastenkombinationen so angeben wie man sie selber drücken würde, da RapidBATCH hier die Eingabe eines Benutzers simuliert, d.h. so tut, als ob der Benutzer das fernzusteuende Programm bedient.

- Einfache Buchstaben, Zahlen und Satzzeichen (also Wörter oder ganze Texte) werden direkt angegeben, wie z.B. "abc" oder "Hallo Welt!".
- Sondertasten wie STRG, ALT, SHIFT, ENTER, etc. werden geklammert und mit der englischen Bezeichnung angegeben. Wird z.B. das Kommando "Zeile 1(enter)Zeile 2" per SENDKEYS an eine Anwendung gesendet (z.B. einen Texteditor), so wird ein Zeilenumbruch zwischen "Zeile 1" und "Zeile 2" erzeugt.
- Um eine Taste gedrückt zu halten, wird ihr ein Zirkumflexzeichen (^) vorangestellt. Das gedrückt halten von Tasten ist bei Sondertasten, z.B. STRG oder ALT in Verbindung mit anderen Tasten möglich. Um mehrere Tasten gleichzeitig zu drücken, muss das Zirkumflexzeichen vor jedes der gewünschten Zeichen geschrieben werden. Das drücken wird automatisch aufgehoben, wenn ein nicht-gedrücktes Zeichen folgt oder der Tastaturkommando-String endet. Beispiel: "^(\alt)f" sendet ein ALT+F, "^(alt)^(strg)^(del)" drückt ALT+STRG+ENTF gleichzeitig, "^(alt)(f1)" sendet ein ALT+F1. "a" würde ein gedrückt halten der Taste "a" hervorrufen. Beachten Sie bitte, daß alle gedrückten Tasten nach der Abarbeitung der Anweisung SENDKEYS wieder automatisch "losgelassen" werden.
- Möchte man ein Zeichen der Makrosprache wie z.B. eine Klammer auf oder Klammer zu senden, stellt man ihr ein Tilde-Zeichen (~) voran. So z.B. ~( für ein "("). Mit ~^ wird ein Zirkumflexzeichen selber gesendet. Um ein einzelnes Tilde-Zeichen zu senden muss ein ~~ angegeben werden.

Als Sondertasten können folgende Tasten angegeben werden.

(tab)	Tabulator-Taste
(clear)	Entfernen-Taste
(enter) oder (return)	Enter-Taste
(shift)	Umschalt-Taste
(ctrl) oder (control)	Strg-Taste
(alt)	Alt-Taste
(pause)	Pause-Taste
(capslock)	Capslock-Taste
(esc)	Escape-Taste
(space)	Leertaste
(pageup)	Bild-hoch Taste
(pagedown)	Bild-runter Taste
(end)	Zeilenende
(home)	Zeilenanfang
(left)	Pfeiltaste Links
(up)	Pfeiltaste hoch
(right)	Pfeiltaste rechts
(down)	Pfeiltaste runter
(select)	Auswahl-Taste
(print)	Druck-Taste
(snapshot)	Bildschirm-Schnappschuss
(insert)	Einfügen-Taste
(delete)	Entfernen-Taste
(help)	Hilfe-taste
(lwin)	Linke Windows-Taste
(rwin)	Rechte Windows-Taste
(apps)	Anwendungs-Taste
(f1)	F1-Taste
(f2)	F2-Taste
(f3)	F3-Taste

(f4)	F4-Taste
(f5)	F5-Taste
(f6)	F6-Taste
(f7)	F7-Taste
(f8)	F8-Taste
(f9)	F9-Taste
(f10)	F10-Taste
(f11)	F11-Taste
(f12)	F12-Taste
(f13)	F13-Taste
(f14)	F14-Taste
(f15)	F15-Taste
(f16)	F16-Taste
(numlock)	NumLock-Taste
(scroll)	Scroll-Taste
(lshift)	Linke Umschalt-Taste
(rshift)	Rechte Umschalt-Taste
(lcontrol)	Linke Strg-Taste
(rcontrol)	Rechte Strg-Taste
(lalt)	Linke Alt-Taste
(ralt)	Rechte Alt-Taste

Als Fenstertitel können auch die Prozess-ID einer Windows-Anwendung oder die build-in Konstanten "DESKTOP" (sendet Tastenanschläge an den Desktop), "SHELL" (sendet Tastenanschläge an die Shell) und "FOREGROUND" (sendet Tastenanschläge an das aktuelle Vordergrundfenster) übergeben werden.

**Beispiele:**

```
rem Notepad starten
shell [windir] # '\notepad.exe', 'show'

rem Tastenkombinationen senden
sendkeys 'Unbenannt - Editor', 'Hello World^(alt)fstest.txt(enter)'
end
```

**Bemerkungen:**

[errorcode] erhält -1, wenn ein Fehler auftrat (z.B. wenn das Fenster nicht gefunden wurde), ansonsten 0.

## MOUSEEVENT

**Syntax:**

MOUSEEVENT Ereignis-Befehl, Positionsangabe

**Beschreibung:**

Simuliert Mauseaktionen.

Die Windows-Maus wird dabei von RapidBATCH ferngesteuert, es kann also eine komplette Automatisierung von Mausebewegungen, Klicks und anderen Mauseignissen erfolgen.

Als Parameter werden ein Event-Kommando sowie die jeweilige X- und Y-Koordinate (in Pixel) als Stringausdruck im Format "X-Position|Y-Position" erwartet (wobei das Pipe-Zeichen das als Standard-Listenseparator definierte Zeichen



repräsentiert). Dieses Event-Kommando wird durch eine Zahl oder die jeweilige build-in Konstante repräsentiert, und simuliert Maustasteneignisse.

- '0' oder MOVE (Maus nur bewegen, kein Tastendruck)
- '1' oder CLICK\_LEFT (einfachen Linksklick simulieren)
- '2' oder DBLCLICK\_LEFT (doppelten Linksklick simulieren)
- '3' oder CLICK\_RIGHT (einfachen Rechtsklick simulieren)
- '4' oder DBLCLICK\_RIGHT (doppelten Rechtsklick simulieren)
- '5' oder DOWN\_LEFT (Linke Maustaste gedrückt halten)
- '6' oder UP\_LEFT (Linke Maustaste loslassen)
- '7' oder DOWN\_RIGHT (Rechte Maustaste gedrückt halten)
- '8' oder UP\_RIGHT (Rechte Maustaste loslassen)

## Beispiele:

```
infobox 'MouseEvent-Demo', 'show'
wait '1000'

rem Prozedur zur Anzeige der X/Y-Koordinaten
proc viewpos
    infobox 'X: ' # [mouse_x] # ' Y: ' # [mouse_y], 'show'
    wait '1'
endproc

rem Bewege Mauszeiger horizontal
[i] = [ScreenWidth]
repeat
    mouseevent 'move', [i] # '|'
    viewpos
    [i] - '1'
until [i] = '0'

rem Bewege Mauszeiger vertikal
[i] = '0'
repeat
    [i] + '1'
    mouseevent 'move', '1|' # [i]
    viewpos
until [i] = [ScreenHeight]

rem Bewege Mauszeiger diagonal
[i] = '0'
repeat
    [i] + '1'
    mouseevent 'move', [i] # '|' # [i]
    viewpos
until [i] = [ScreenWidth]
end
```

## Bemerkungen:

Verwenden Sie die SENDKEYS-Anweisung, um Tastenanschläge zu simulieren.

Alternativ können auch die vordefinierten Variablen [mouse\_x] und [mouse\_y] dazu verwendet werden, die Mausposition zu verändern.

## LOCKWORKSTATION

### Syntax:

LOCKWORKSTATION

**Beschreibung:**

Sperrt eine Windows 2000 oder Windows XP Professional Workstation.

Diese Anweisung ist unter Windows 95/98/ME/NT sowie XP Home unwirksam.

**Beispiele:**

```
rem Arbeitsstation sperren?
confirm [sperren] = 'Computer sperren?'
if [sperren] = [true]
    rem Computer sperren!
    lockworkstation
endif
end
```

**Bemerkungen:**

[errorcode] erhält den Wert -1, wenn die Arbeitsstation nicht gesperrt werden konnte (z.B. wenn sie schon gesperrt ist oder das Betriebssystem diese Funktion nicht unterstützt). Wenn keine Fehler auftraten erhält [errorcode] den Wert 0.

## SHUTDOWN

**Syntax:**

SHUTDOWN Modus

**Beschreibung:**

Beendet Windows und rebootet oder fährt den PC herunter.

Als Modus muss einer der folgenden numerischen Werte oder Build-in Konstanten angegeben werden:

0 (SHUTDOWN) - PC herunterfahren

1 (REBOOT) - PC rebooten

2 (POWEROFF) - PC herunterfahren und ausschalten (falls dies Hardware-bedingt möglich ist)

3 (LOGOFF) - Aktuellen Benutzer ausloggen und Anmeldebildschirm anzeigen

**Beispiele:**

```
echo 'Restarting computer...'
shutdown 'reboot'
end
```

**Bemerkungen:**

[errorcode] erhält im Fehlerfall den Wert -1, ansonsten 0.

## Benutzerdefinierte Dialoge

### NEWDIALOG

#### Syntax:

NEWDIALOG Widget-Label, Widget-Typ, Startkoordinaten

#### Beschreibung:

Erzeugt ein neues Dialog-Widget.

NEWDIALOG kann sowohl dazu verwendet werden, ein Dialog-Basisfenster zu erzeugen als auch ein Dialog-Childwidget wie z.B. Buttons oder Eingabefelder. NEWDIALOG benötigt als Parameter ein Widget-Label, unter dem das Widget später angesprochen werden kann, den Widget-Typ (siehe unten) sowie die Widget-Startposition in der Form "X|Y|Höhe|Breite", wobei all diese Angaben in Pixel anzugeben sind. Die Separierung der einzelnen Werte erfolgt durch das in [std\_sep] definierte Separationszeichen (standardmäßig das Pipe-Zeichen "|"). Die X- und Y-Koordinaten sind nur bei Dialogfenstern relativ zur linken-oberen Ecke des Anzeigebildschirms. Bei allen Widgets, die auf einem Dialogfenster erzeugt werden, sind die X- und Y-Koordinaten relativ zur linken-oberen Ecke des jeweiligen Basisdialogs.

Das Widget-Label muss bei allen Childelementen mit dem Label des Basisdialogs, auf dem das Childelement erzeugt werden soll, gefolgt von einem Doppelpunkt (:), beginnen. Ein Widget-Label für die Erzeugung eines Childelements auf dem Dialog "Basisdialog" wäre zum Beispiel: "Basisdialog:Childelement".

Folgend eine Liste an möglichen Werten, die bei NEWDIALOG als Dialogelement-Typ angegeben werden können.

Widget-Typ:	Beschreibung:
DIALOG	Dialogfenster
BUTTON	Button mit Text
IMAGEBUTTON	Button mit Bild
INPUT	einzeiliges Eingabefeld
PWD	einzeiliges Passwort-Eingabefeld
EDIT	mehrzeiliges Eingabefeld
EDIT_LINEWRAP	mehrzeiliges Eingabefeld mit automatischem Zeilenumbruch
LABEL	statisches Textlabel
OPTION	Optionsfeld
RADIO	Radiobutton
LIST	Listenelement
LIST_SORTED	Sortiertes Listenelement
LIST_MULTI	Listenelement mit Mehrfachauswahl
LIST_MULTISORTED	Sortiertes Listenelement mit Mehrfachauswahl
PROGRESS	Fortschrittsbalken (Progressbar)
COMBO	Combobox
STATIC_COMBO	Combobox mit festen Werten
IMAGE	statisches Bitmap
GROUP	statische Groupbox

#### Beispiele:

```
rem Erzeugung eines Dialogfensters
newdialog 'myDialog', 'DIALOG', '200|200|500|250'

rem Erzeugung eines Buttons auf dem oben erzeugten Dialogfenster
newdialog 'myDialog:myButton', 'BUTTON', '1|1|100|25'
```

#### Bemerkungen:

Schlägt die Erzeugung eines Widgets fehl, erhält [errorcode] den Wert -1, ansonsten 0.

Verwenden Sie die Anweisung LETDIALOG, um Eigenschaften eines Widgets zu verändern. Verwenden Sie gegenteilig zu LETDIALOG die Funktion GETDIALOG, um Eigenschaften eines Widgets zu ermitteln.

Mit Hilfe der RUNDIALOG-Funktion lässt sich auf Ereignisse, die mit oder auf dem Widget auftreten können (z.B. anklicken eines Buttons durch den Benutzer), warten und anschließend anhand des zurückgelieferten Event-Strings reagieren.

Bis auf Dialogfenster werden alle Widgets sofort sichtbar geschaltet. Basisdialoge müssen vor der ersten Ausführung von RUNDIALOG sichtbar gemacht werden (mit Hilfe der Anweisung LETDIALOG und der Dialogeigenschaft "VISIBLE").

Wird ein Dialogfenster innerhalb einer Library-Funktion erzeugt, sollten seine Ressourcen nach Beendigung des Dialoges mit der Anweisung DELDIALOG wieder freigegeben werden.

## LETDIALOG

### Syntax:

LETDIALOG Widget-Label, Attributsbezeichner, Attributparameter

### Beschreibung:

LETDIALOG ermöglicht das Setzen verschiedenster Widget-Attribute und -Ereignisse.

Die Anweisung erwartet neben dem Widget-Label, über das das Widget identifiziert wird, einen so genannten Attribut-Bezeichner für das Attribut, das gesetzt oder geändert werden soll, und einen entsprechenden Wert oder Attributparameter, der für das jeweilige Attribut gesetzt werden soll. Dieser Wert ist immer abhängig vom jeweiligen Attribut, welches angegeben wurde.

Die folgende Auflistung beschreibt die verschiedenen Attribute, erklärt die möglichen Attributparameter-Werte und gibt Auskunft über die unterstützten Widgets für jedes Attribut (kein Widget unterstützt alle möglichen Attribute!).

CAPTION	Setzt den Titel eines Widgets.			
Widgets	DIALOG	BUTTON	LABEL	RADIO
	OPTION	GROUP	MENU	
Parameterwerte	Ein beliebiger Text.			

CHECKED	Status, ob das Widget "gehakt" ist.			
Widgets	RADIO	OPTION	MENU	
Parameterwerte	0 = gehakt	-1 = nicht gehakt		

ENABLED	Ermöglicht das (De-)Aktivieren von Widgets. Deaktivierte Widgets sind zwar sichtbar, jedoch "ausgegraut" und nicht benutzbar.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LABEL	OPTION
	RADIO	GROUP	COMBO	STATIC_COMBO
	LIST	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED
	PWD	MENU	PROGRESS	
Parameterwerte	0 = gehakt	-1 = nicht gehakt		

FOCUS	Setzt den Eingabefokus auf das Dialogelement.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	OPTION	RADIO
	COMBO	STATIC_COMBO	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	PWD	
Parameterwerte	Nicht erforderlich, Leerstring			

FONT	Setzt eine Schriftart für ein Widget. Neben den Schriftart-Namen wird die Größe in Punkt sowie ein Style-Wert übergeben. Dieser Style-Wert ermöglicht die Darstellung der Schrift in Fett (2), Kursiv (4) oder Unterstrichen (8). Rechnet man die jeweiligen Werte (2, 4, 8) zusammen, kann man mehrere Styles gleichzeitig setzen (z.B. Wert 6 für Fett und Kursiv).			
Widgets	BUTTON	INPUT	EDIT	EDIT_LINEWRAP
	LABEL	OPTION	RADIO	GROUP
	COMBO	STATIC_COMBO	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	PWD	
Parameterwerte	Schriftart-Typ, Größe und Style in der Form "Schriftart Größe Style", z.B. "Courier New 10 2"			

HEIGHT	Höhe eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	Die Widget-Höhe in Pixel.			

ICON	Setzt ein alternatives Icon für einen Dialog. Wird ICON auf leer gesetzt (default), wird das globale Anwendungsicon, welches in der Variablen [app_icon] definiert ist, verwendet.			
Widgets	DIALOG			
Parameterwerte	Relativer oder absoluter Pfad zu einer externen ICO-Datei.			

IMAGE	Setzt eine externe Bitmapdatei (Dateiendung: BMP) für ein Bild-anzeigendes Widget.			
Widgets	IMAGEBUTTON	IMAGE		
Parameterwerte	Relativer oder absoluter Pfad zu einer Bitmap-Datei.			

IMAGE_TRANSPARENT	Setzt eine externe Bitmapdatei (Dateiendung: BMP) für ein Bild-anzeigendes Widget. Hierbei wird das Farbpixel der linken oberen Ecke des Bitmaps als Transparenzfarbe verwendet und damit transparent auf dem Widget dargestellt. Es können nur Bilder mit einer maximalen Farbtiefe von 8 BPP transparent dargestellt werden (Windows-Einschränkung).			
Widgets	IMAGEBUTTON	IMAGE		
Parameterwerte	Relativer oder absoluter Pfad zu einer Bitmap-Datei.			

ITEMS	Liste an Werten in Dialog-Listenelementen.			
Widgets	LIST	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED
	COMBO	STATIC_COMBO		
Parameterwerte	Eine Liste an Elementen, in der jedes Element durch das Standard-Separationszeichen (standardmäßig Pipe " ") separiert ist. Beispiel: "Item 1 Item 2 Item 3"			

MENU	<p>Erzeugt ein Menü auf einem Dialogfenster.  Dieses Menü wird als Beschreibung über einen so genannten Menu-Descriptor übergeben.  In diesem Descriptor beginnt jeder neue Menüeintrag auf einer Ebene mit einem Label, z.B. "Menü-1". Dann folgt entweder ein Doppelpunkt, der ein neues Untermenü einleitet. Folgt ein Pipe-Zeichen ( ) (bzw. das in [std_sep] definierte Zeichen) wird ein Menü in der selben Ebene neu angelegt. Ein Untermenü muss mit einem Semikolon (;) abgeschlossen werden.  Für jeden "anklickbaren" Menüpunkt (also jeder Menüpunkt ohne Untermenü) wird als Dialog-Element ein Label in der Form "Fenstername:Menu_Menüeintrag" erzeugt.  Über dieses Dialogelement können dann auch Attribute wie CAPTION, CHECKED oder ENABLED für einen Menüpunkt gesetzt werden.  Wird das MENU-Attribut eines Basisdialogs auf leer gesetzt, wird das Menü entfernt.</p>			
Widgets	DIALOG			
Parameterwerte	Menu-Descriptor			

ORDER	<p>Setzt die Z-Order (Bildschirmordnung) eines Dialogfensters.  Hierbei gibt es folgende Modi:</p> <ul style="list-style-type: none"> <li>• TOP → Das Fenster wird vor allen sichtbaren und aktiven Fenstern geschoben und kann nicht in den Hintergrund gebracht werden! Diese Option wird auch vom build-in Dialog INFOBOX verwendet! Dieser Modus kann durch setzen auf NORMAL wieder verlassen werden.</li> <li>• BOTTOM → Das Dialogfenster wird als letztes Fenster hinter allen sichtbaren Anwendungsfenstern geschoben. Dieser Modus kann durch setzen auf NORMAL wieder verlassen werden.</li> <li>• NORMAL (Standard) → Das Fenster wird in den Normalmodus versetzt, und kann in den Vorder- bzw. Hintergrund gebracht werden.</li> </ul>			
Widgets	DIALOG			
Parameterwerte	"NORMAL"	"BOTTOM"	"TOP"	

POSITION	<p>Setzt die Position eines Widgets. Bei Basisdialogen sind X-/Y-Koordinaten relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Basisdialogs, zu dem das Widget gehört.</p>			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	PWD	EDIT	EDIT_LINEWRAP	LABEL
	OPTION	RADIO	GROUP	IMAGE
	COMBO	STATIC_COMBO	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	PROGRESS	
Parameterwerte	Die Angabe der Koordinaten erfolgt in Pixeln.			

PROGRESS	Ändert den Fortschrittsbalken einer Fortschrittsanzeige (Progressbar).			
Widgets	PROGRESS			
Parameterwerte	Ein ganzzahliger, prozentualer Fortschrittswert (Minimalwert 0, Maximalwert 100).			

READONLY	Schaltet den "Nur lesen" Modus von Eingabefeldern ein oder aus.			
Widgets	EDIT	EDIT_LINEWRAP	INPUT, PWD	
Parameterwerte	0 = "nur lesen"	-1 = "schreibbar" (Standard)		

SELECTION	<p>Bei Eingabefeldern: Ersetzt den markierten Text durch den als Attributparameter übergebenen Wert. Ist kein Text markiert, wird der übergebene Text einfach an der Cursorposition eingefügt.</p> <p>Bei Listefeldern: Setzt die Auswahl im Listefeld auf das nächst-passende Element.</p>			
Widgets	INPUT	EDIT	EDIT_LINEWRAP	LIST
	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED	PWD
	STATIC_COMBO			
Parameterwerte	Zu ersetzender/einzufügender Text bzw. zu markierender Eintrag (bei LIST-Widget)			

SELINDEX	Setzt die Selektion von Listenauswahlfeldern anhand des Eintrag-Indexes, der selektiert werden soll.			
Widgets	LIST	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED
Parameterwerte	Bei Listenauswahlfeldern mit einfacher Auswahl der Index des zu selektierenden Eintrags, bei Listenauswahlfeldern mit Mehrfachauswahl eine Liste, getrennt durch Pipes, mit den zu selektierenden Indizes.			

SELPOS	Setzt den Cursor in einem Eingabefeld auf eine bestimmte Position oder selektiert Text.			
Widgets	INPUT	EDIT	EDIT_LINEWRAP	PWD
	COMBO			
Parameterwerte	Markierung/Cursorposition in der Form "Cursorposition Markierungslänge", z.B. "1 10" markiert die ersten 10 Zeichen, "15 0" setzt den Cursor auf Zeichen 15 (ohne Markierung).			

STATUSBAR	<p>Erzeugt eine Statusbar auf einem Dialogfenster.</p> <p>Statusbars können Statusinformationen zum Programm annehmen und in mehrere Bereiche abgeteilt werden. Wird nur ein Bereich benötigt, wird Statusbar auf einen entsprechenden String-Wert gesetzt; Sollen mehrere Bereiche erzeugt werden, so wird eine Liste (separiert durch Pipe-Zeichen " ") in folgendem Format übergeben:</p> <p>Text im Bereich Größe des Bereichs beginnend vom linken Rand des Dialogfensters in Pixel</p> <p>Soll nun ein weiterer Bereich folgen, wird dieser einfach hinter den ersten gehängt.</p> <p>Wird das STATUSBAR-Attribut eines Basisdialoges auf leer gesetzt, wird die Statusbar vom Dialog entfernt.</p>			
Widgets	DIALOG			
Parameterwerte	Statusbar-Descriptor			

STYLE	<p>Setzt ein Style für ein Dialogfenster</p> <p>Hierbei gibt es folgende Styles:</p> <ul style="list-style-type: none"> <li>• STANDARD à Ein Dialogfenster mit Schließen-, Minimieren- und Maximieren-Button sowie mit Systemmenü</li> <li>• DIALOG à Ein Dialogfenster mit Schließen-Button und Systemmenü</li> <li>• SINGLE à Ein Dialogfenster mit Schließen-Button, Minimieren-Button und Systemmenü</li> <li>• TOOL à Ein Dialogfenster im Toolwindow-Stil mit Schließen-Button, skalierbar</li> <li>• STATIC_TOOL à Ein Dialogfenster im Toolwindow-Stil, nicht skalierbar (statische Größe)</li> </ul>			
Widgets	DIALOG			
Parameterwerte	STANDARD	DIALOG	SINGLE	TOOL
	STATIC_TOOL			

TEXT	Setzt den Textbereich eines Eingabe-Dialogelements			
Widgets	EDIT	EDIT_LINEWRAP	INPUT	PWD
	COMBO			
Parameterwerte	Ein beliebiger Text.			

TOOLTIP	Setzt einen Tooltip für das jeweilige Widget. Tooltips sind Hilfstexte, die aufgeblendet werden, wenn der Benutzer mit der Maus über ein Widget fährt.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	Ein beliebiger Hilfstext.			

<b>VISIBLE</b>	Setzt die Sichtbarkeit eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	0 = sichtbar	-1 = unsichtbar		

<b>WIDTH</b>	Breite eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	Die Widget-Breite in Pixel.			

<b>X</b>	Die X-Position eines Widgets. Bei Basisdialogen ist diese Koordinate relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Basisdialogs, zu dem das Widget gehört.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	Die X-Position des Widgets, in Pixel.			

<b>Y</b>	Die Y-Position eines Widgets. Bei Basisdialogen ist diese Koordinate relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Basisdialogs, zu dem das Widget gehört.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Parameterwerte	Die Y-Position des Widgets, in Pixel.			

Mit LETDIALOG lassen sich neben Attributen auch Events (also Ereignisse, die bei der Ausführung von RUNDIALOG auftreten können) für die verschiedenen Dialogelemente ein- und ausschalten. Die folgende Tabelle beschreibt und erklärt die verschiedenen Ereignisse und gibt Auskunft über die unterstützten Dialogelemente für jedes Ereignis (kein Dialogelement unterstützt alle möglichen Ereignisse!).

Das jeweilige Ereignis kann durch Setzen auf den Wert 0 eingeschaltet und durch Setzen auf den Wert -1 ausgeschaltet werden. Der Default-Wert für die verschiedenen Ereignisse wird hinter den unterstützten Elementen in Klammern angegeben.

Ereignis-Bezeichner:	Beschreibung:	Unterstützte Widgets:
EVENT_CLICK	Ereignis beim Anklicken eines Widgets.	DIALOG (-1), BUTTON (0), IMAGEBUTTON (0), LABEL (-1), OPTION (-1), RADIO (-1), IMAGE (-1), MENU (0)
EVENT_DBLCLICK	Ereignis beim doppelten Anklicken eines Widgets.	LIST (-1), LIST_SORTED (-1)
EVENT_CHANGE	Ereignis beim Ändern eines Eingabefeldes.	INPUT (-1), EDIT (-1), EDIT_LINEWRAP (-1), PWD (-1), COMBO (-1), STATIC_COMBO (-1)



EVENT_CLOSE	Ereignis beim Schließen eines Dialoges.	DIALOG (0)
EVENT_SIZE	Ereignis beim Vergrößern/Verkleinern eines Dialoges.	DIALOG (0)
EVENT_MOVE	Ereignis beim Verschieben eines Dialoges.	DIALOG (-1)
EVENT_MOUSEMOVE	Ereignis beim Bewegen der Maus auf einem Dialogfenster.	DIALOG (-1)
EVENT_OK	Ereignis beim Drücken von Enter auf einem Dialog.	DIALOG (-1)
EVENT_CANCEL	Ereignis beim Drücken von Esc auf einem Dialog.	DIALOG (-1)

## Beispiele:

```

rem ACHTUNG:
rem Dieses Beispiel ist nicht lauffähig, da die Widgets, die hier
rem angesprochen werden, noch nicht erzeugt worden sind.
rem Das Beispiel soll lediglich demonstrieren, wie bestimmte
rem Attribute auf Widgets gesetzt und übergeben werden.

rem Setzen/Ändern der Widget-Position und -Größe
letdialog 'myDialog', 'POSITION', '1|1|800|600'
letdialog 'myDialog:Button1', 'POSITION', '10|10|100|25'

rem Setzen von Überschriften/Beschriftungen
letdialog 'myDialog', 'CAPTION', 'Dies ist ein Dialogtitel'
letdialog 'myDialog:Button1', 'CAPTION', '&Ok'

rem Setzen von Textinhalten in Eingabefeldern
letdialog 'myDialog:Eingabe', 'TEXT', 'Diesen Text können Sie editieren!'
letdialog 'myDialog:Editor', 'TEXT', 'Dieser Text hier ist' # [new_line] # 'mehrzeilig!'

rem Füllen der Listen von LIST-, LIST_SORTED- und COMBO-Elementen
letdialog 'myDialog:Liste', 'ITEMS', 'Montag|Dienstag|Mittwoch|Donnerstag|Freitag'
letdialog 'myDialog:Combo', 'ITEMS', 'Samstag|Sonntag'

rem Häkchen bei OPTION-, RADIO- und MENU-Elementen ein-/ausschalten
letdialog 'myDialog:Option1', 'CHECKED', '0'
letdialog 'myDialog:Radio1', 'CHECKED', '-1'
letdialog 'myDialog:Menu_&Nichts machen', 'CHECKED', '0'

rem Eingabefokus setzen
letdialog 'myDialog:Eingabe', 'FOCUS', ''

rem Dialogelement aktivieren/deaktivieren
letdialog 'myDialog:Eingabe', 'ENABLED', '-1'
letdialog 'myDialog:Eingabe', 'ENABLED', '0'

rem Tooltip setzen
letdialog 'myDialog:Button1', 'TOOLTIP', 'Drücken Sie hier drauf!'

rem Setzen/Ersetzen von markiertem Text bzw. Eintrag aus LIST-Element
letdialog 'myDialog:Eingabe', 'SELECTION', 'Neuer Text!'
letdialog 'myDialog:Editor', 'SELECTION', [clipboard]
letdialog 'myDialog:Liste', 'SELECTION', 'Dienstag'

rem Cursorposition/Markierung setzen
letdialog 'myDialog:Eingabe', 'SELPOS', '1|1'
letdialog 'myDialog:Editor', 'SELPOS', '10|100'

rem Schriftart setzen
letdialog 'myDialog:Eingabe', 'FONT', 'Courier New|12|0'
letdialog 'myDialog:Button1', 'FONT', 'Arial Black|10|2'

rem Sichtbarkeit ein-/ausschalten
letdialog 'myDialog', 'VISIBLE', '0'

```

```
letdialog 'myDialog:Button1', 'VISIBLE', '-1'

rem Menü auf einem Dialogfenster erzeugen
letdialog 'myDialog', 'MENU', '&Datei:&Neu|Ö&ffnen|-|&Beenden;'

rem Statusbars auf einem Dialogfenster erzeugen
letdialog 'myDialog', 'STATUSBAR', 'Eine einfache Statusbar'

[status] = 'Diese Statusbar|300'
[status] # '|ist in mehrere Bereiche|500'
[status] # '|unterteilt!|' # [ScreenWidth]
letdialog 'myDialog', 'STATUSBAR', [status]

rem Ausschalten von Ereignissen
letdialog 'myDialog:Button1', 'EVENT_CLICK', '-1'
letdialog 'myDialog', 'EVENT_CLOSE', [false]

rem Einschalten von Ereignissen
letdialog 'myDialog:Eingabe', 'EVENT_CHANGE', '0'
letdialog 'myDialog', 'EVENT_CLOSE', [true]
```

## Bemerkungen:

Im Fehlerfall (z.B. wenn das angegebene Widget nicht gefunden wurde) erhält die Variable [errorcode] den Wert -1, ansonsten 0.

Zum Auslesen der bei LETDIALOG gesetzten Werte sowie z.B. vom Benutzer veränderter Werte (z.B. in einem Editfeld) wird die Funktion GETDIALOG verwendet.

Die Anweisung NEWDIALOG ermöglicht das Erzeugen von Widgets.

## GETDIALOG

### Syntax:

GETDIALOG Attributswert = Widget-Label, Attributsbezeichner

### Beschreibung:

Gibt den Wert eines Widget-Attributs zurück.

Die Funktion erwartet neben dem Widget-Label, über welches das Dialogelement angesprochen wird, einen so genannten Attribut-Bezeichner, dessen entsprechender Wert zurückgeliefert wird. Der zurückgegebene Wert ist also immer abhängig vom jeweiligen Attribut.

Die folgende Tabelle beschreibt die verschiedenen Attribute und gibt Auskunft über den zurückgegebenen Wert (sowie evtl. Formatierungen etc.).

CAPTION	Der Titel eines Widgets.			
Widgets	DIALOG	BUTTON	LABEL	RADIO
	OPTION	GROUP	MENU	
Rückgabewert	Der Titel des entsprechenden Widgets.			

CHECKED	Status, ob das Widget "gehakt" ist.			
Widgets	RADIO	OPTION	MENU	
Rückgabewert	0 = gehakt	-1 = nicht gehakt		

ENABLED	Ermittelt, ob ein Dialogelement aktiviert oder deaktiviert ist. Deaktivierte Dialogelemente sind zwar sichtbar, jedoch "ausgegraut" und nicht benutzbar.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	LABEL	OPTION	RADIO
	GROUP	COMBO	STATIC_COMBO	LIST
	LIST_SORTED	PWD	MENU	PROGRESS
Rückgabewert	0 = aktiviert		-1 = deaktiviert (ausgegraut)	

FOCUS	Ermittelt, ob ein Dialogelement den Eingabefokus hat oder nicht.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	OPTION	RADIO
	COMBO	STATIC_COMBO	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	PWD	
Rückgabewert	0 = Checked		-1 = Unchecked	

FONT	Gibt die Schriftartdefinition eines Widgets zurück.			
Widgets	BUTTON	INPUT	PWD	EDIT
	EDIT_LINEWRAP	LABEL	OPTION	RADIO
	GROUP	COMBO	STATIC_COMBO	LIST
	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED	
Rückgabewert	<p>Schriftartdefinition im Format "Schriftart-Name Schriftgröße Schriftstil"</p> <p>Wurde dem Widget zuvor keine benutzerdefinierte Schriftart zugewiesen, wird ein Leerstring zurückgegeben.</p>			

HEIGHT	Ermittelt die Höhe eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Die Widget-Höhe in Pixel.			

HWND	Gibt das interne Windows-Handle (von Typ HWND (long)) eines Widgets zurück, so daß dieses aus einer DLL, die mit der EXT-Anweisung aufgerufen wird, angesprochen werden kann. Dieser Wert ist nur für Entwickler interessant, die RapidBATCH DLLs programmieren.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Der HWND-Wert des jeweiligen Widgets.			

ICON	Alternatives Icon eines Dialogfensters.			
Widgets	DIALOG			
Rückgabewert	Relativer oder absoluter Pfad der ICO-Datei, wenn gesetzt. Wurde dem Widget kein Icon zuge- teilt, wird ein Leerstring zurückgeliefert.			

IMAGE	Pfad der Bitmap-Datei eines Bild-anzeigenden Widgets.			
Widgets	IMAGEBUTTON	IMAGE		
Rückgabewert	Relativer oder absoluter Pfad der BMP-Datei.			

IMAGE_TRANSPARENT	Pfad der transparenten Bitmap-Datei eines Bild-anzeigenden Widgets.			
Widgets	IMAGEBUTTON	IMAGE		
Rückgabewert	Relativer oder absoluter Pfad der BMP-Datei.			

ITEMS	Liste an Werten in Listenauswahlfeldern.			
Widgets	LIST	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED
	COMBO	STATIC_COMBO		
Rückgabewert	Liste an Elementen, in der jedes Element durch das Standard-Separationszeichen (standardmäßig Pipe " ") separiert ist. Beispiel: "Item 1 Item 2 Item 3"			

MENU	Ermittelt den Menüdescriptor eines Dialogfensters.			
Widgets	DIALOG			
Rückgabewert	Menu Descriptor, wie er übergeben wurde, oder Leerstring, wenn dem Dialogfenster kein Menü zugewiesen wurde.			

ORDER	Dialog-Anordnung (Z-Order).			
Widgets	DIALOG			
Rückgabewert	Der aktuelle Anordnungsmodus des Dialogfensters auf der Z-Achse. Mögliche Werte sind "BOTTOM", "TOP" oder ein Leerstring, der für den Modus "NORMAL" steht.			

POSITION	Die Position eines Widgets. Bei Basisdialogen sind die entsprechenden X/Y-Koordinaten relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Dialogfensters, zu dem das Child-Widget jeweils gehört.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	PWD	EDIT	EDIT_LINEWRAP	LABEL
	OPTION	RADIO	GROUP	IMAGE
	COMBO	STATIC_COMBO	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	PROGRESS	
Rückgabewert	Die Position des Widgets in der Form "X Y Breite Höhe". Die Koordinatenwerte liegen in Bildschirmpixeln vor.			

PROGRESS	Gibt den aktuellen Fortschritt eines Fortschrittsbalken-Widgets (Progressbar) zurück.			
Widgets	PROGRESS			
Rückgabewert	Ein prozentualer Wert, den die Progressbar derzeit repräsentiert (Minimalwert 0, Maximalwert 100).			

READONLY	Ermittelt, ob ein Texteingabefeld auf "nur lesen" gesetzt ist. In diesen Feldern kann kein Wert eingegeben oder geändert werden, Text kann jedoch markiert werden.			
Widgets	EDIT	EDIT_LINEWRAP	INPUT	PWD
Rückgabewert	0 = Nur-lesen Modus aktiviert		-1 = Nur-lesen Modus deaktiviert (Standard)	

<b>SELECTION</b>	Aktuelle Auswahl in einem Eingabe- oder Listenfeld.			
Widgets	INPUT	EDIT	EDIT_LINEWRAP	LIST
	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED	PWD
	COMBO	STATIC_COMBO		
Rückgabewert	<p>Bei Eingabefeldern: Der momentan markierte Text.</p> <p>Bei Listenfeldern: Das selektierte Element bzw. eine Liste der selektierten Elemente, jeweils durch Pipe-Zeichen getrennt.</p>			

<b>SELINDEX</b>	Aktuelle Auswahl in Listenfeldern.			
Widgets	LIST	LIST_SORTED	LIST_MULTI	LIST_MULTISORTED
Rückgabewert	Der Index des gewählten Eintrags bzw. eine Liste von Indizes (jeweils getrennt durch Pipe-Zeichen ( )) der gewählten Einträge bei Mehrfachauswahl.			

<b>SELPOS</b>	Position des Cursors in einem Eingabewidget und Länge der Markierung.			
Widgets	INPUT	EDIT	EDIT_LINEWRAP	PWD
	COMBO			
Rückgabewert	Markierung/Cursorposition in der Form "Cursorposition Markierungslänge", z.B. "1 10".			

<b>STATUSBAR</b>	Statusbar-Descriptor eines Dialogfensters.			
Widgets	DIALOG			
Rückgabewert	Der Statusbar-Descriptor, der bei LETDIALOG gesetzt wurde, oder ein Leerstring, wenn keine Statusbar existiert.			

<b>STYLE</b>	Dialog-Style			
Widgets	DIALOG			
Rückgabewert	Der Style-Wert. Wurde dem Widget kein Style zugewiesen (Default-Style, STANDARD) wird ein Leerstring zurückgegeben.			

<b>TEXT</b>	Inhalt des Textbereichs eines Eingabe-Widgets.			
Widgets	EDIT	EDIT_LINEWRAP	INPUT	PWD
	COMBO			
Rückgabewert	Der Text des entsprechenden Widgets.			

<b>TOOLTIP</b>	Hilfstext des Tooltips eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	PWD	EDIT	LABEL	OPTION
	RADIO	GROUP	IMAGE	COMBO
	STATIC_COMBO	LIST	LIST_SORTED	
Rückgabewert	Der Text des entsprechenden Tooltips.			

TYPE	Gibt den Typ eines Widgets als String zurück.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Der Typ des Widgets.			

VISIBLE	Ermittelt, ob ein Widget sichtbar ist oder nicht.			
Widgets	DIALOG	BUTTON	INPUT	EDIT
	LIST	COMBO	STATIC_COMBO	RADIO
	OPTION	GROUP	IMAGE	PWD
Rückgabewert	0 = Sichtbar	-1 = Unsichtbar		

WIDTH	Ermittelt die Breite eines Widgets.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Die Widget-Breite in Pixel.			

X	Ermittelt die X-Position eines Widgets. Bei Basisdialogen ist diese Koordinate relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Basisdialogs, zu dem das Widget gehört.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Die X-Position des Widgets, in Pixel.			

Y	Ermittelt die Y-Position eines Widgets. Bei Basisdialogen ist diese Koordinate relativ zur linken oberen Ecke des Bildschirms, bei Child-Widgets relativ zur linken oberen Ecke des Basisdialogs, zu dem das Widget gehört.			
Widgets	DIALOG	BUTTON	IMAGEBUTTON	INPUT
	EDIT	EDIT_LINEWRAP	LIST	LIST_SORTED
	LIST_MULTI	LIST_MULTISORTED	COMBO	STATIC_COMBO
	RADIO	OPTION	GROUP	IMAGE
	PWD	PROGRESS		
Rückgabewert	Die Y-Position des Widgets, in Pixel.			

Mit GETDIALOG lässt sich auch ermitteln, ob bestimmte Events (Ereignisse, die bei der Ausführung von RUNDIALOG auftreten können) für ein Widget gesetzt sind oder nicht.

Die folgende Tabelle beschreibt und erklärt die verschiedenen Ereignisse und gibt Auskunft über die unterstützten Widgets für jedes Ereignis (kein Widget unterstützt alle möglichen Ereignisse!).

Ob das jeweilige Ereignis gesetzt bzw. eingeschaltet ist, ist an den Werten 0 (= Ereignis ist gesetzt) und -1 (= Ereignis ist nicht gesetzt), die durch GETDIALOG zurückgegeben werden, erkennbar. Der Default-Status, der nach der Erzeugung eines Widgets automatisch für den jeweiligen Element-Typ gesetzt wird, ist in Klammern hinter den unterstützten Dialogelementen angegeben.

Ereignis-Bezeichner:	Beschreibung:	Unterstützte Widgets:
EVENT_CLICK	Ereignis beim Anklicken eines Widgets.	DIALOG (-1), BUTTON (0), IMAGE-BUTTON (0), LABEL (-1), OPTION (-1), RADIO (-1), IMAGE (-1), MENU (0)
EVENT_DBLCLICK	Ereignis beim doppelten Anklicken eines Widgets.	LIST (-1), LIST_SORTED (-1)
EVENT_CHANGE	Ereignis beim Ändern eines Eingabefeldes.	INPUT (-1), EDIT (-1), PWD (-1), COMBO (-1), STATIC_COMBO (-1)
EVENT_CLOSE	Ereignis beim Schließen eines Dialoges.	DIALOG (0)
EVENT_SIZE	Ereignis beim Vergrößern/Verkleinern eines Dialoges.	DIALOG (0)
EVENT_MOVE	Ereignis beim Verschieben eines Dialoges.	DIALOG (-1)
EVENT_MOUSEMOVE	Ereignis beim Bewegen der Maus auf einem Dialogfenster.	DIALOG (-1)
EVENT_OK	Ereignis beim Drücken von Enter auf einem Dialog.	DIALOG (-1)
EVENT_CANCEL	Ereignis beim Drücken von Esc auf einem Dialog.	DIALOG (-1)

## Beispiele:

```
rem ACHTUNG:
rem Dieses Beispiel ist nicht lauffähig, da die Widgets, die hier
rem angesprochen werden, noch nicht erzeugt worden sind.
rem Das Beispiel soll lediglich demonstrieren, wie bestimmte
rem Attribute von Widgets ermittelt und gehandhabt werden.
```

```
rem Ermitteln und extrahieren einer Widget-Position
getdialog [pos] = 'myDialog', 'POSITION'
```

```
gettok [x] = [pos], [std_sep], '1'
gettok [y] = [pos], [std_sep], '2'
gettok [höhe] = [pos], [std_sep], '3'
gettok [breite] = [pos], [std_sep], '4'
```

```
echo 'X: ' # [x] # ' Y: ' # [y]
echo 'Höhe: ' # [höhe] # ' Breite: ' # [breite]
```

```
rem Auslesen des Titels eines Widgets
getdialog [titel] = 'myDialog', 'CAPTION'
echo 'Der Titel lautet: ' # [titel]
```

```
rem Inhalt eines Eingabefeldes auslesen
getdialog [text] = 'myDialog:Eingabel', 'TEXT'
echo 'Der Text lautet: ' # [text]
```

```
rem Alle Einträge eines Listenauswahlfeldes ermitteln
getdialog [einträge] = 'myDialog:Listel', 'ITEMS'
replacevar [einträge] = [einträge], [std_sep], [new_line]
echo [einträge]
```

```
rem Prüfen, ob ein Optionsfeld gehakt ist
getdialog [state] = 'myDialog:Option1', 'CHECKED'
if [state] = [true]
    echo 'Die Option ist eingeschaltet!'
else
    echo 'Die Option ist nicht aktiv.'
endif
```

```
rem Prüfen, ob ein Widget den Eingabefokus hat oder nicht
getdialog [hat_fokus] = 'myDialog:Eingabel', 'FOCUS'
if [hat_fokus] = [true]
    echo 'Das Eingabefeld hat den Fokus!'
else
    echo 'Das Eingabefeld hat keinen Fokus.'
```

```
endif

rem Prüfen ob ein Widget aktiviert oder deaktiviert ist
getdialog [ist_aktiviert] = 'myDialog:Option2', 'ENABLED'
if [ist_aktiviert] = [true]
    echo 'Das Widget ist aktiv und kann verwendet werden.'
else
    echo 'Das Widget ist deaktiviert.'
endif

rem Textauswahl in einem Texteingabefeld
getdialog [seltext] = 'myDialog:Eingabe1', 'SELECTION'
echo 'Der gewählte Text lautet: ' # [seltext]

rem Auslesen einer Mehrfachauswahl
getdialog [einträge] = 'myDialog:Liste2', 'SELECTION'
replacevar [einträge] = [einträge], [std_sep], [new_line]

echo 'Selektierte Einträge:' # [new_line] # [einträge]

rem Auslesen der Cursorposition sowie der Auswahlänge
rem in einem Eingabefeld
getdialog [selpos] = 'myDialog:Eingabe1', 'SELPOS'
gettok [pos] = [selpos], [std_sep], '1'
gettok [len] = [selpos], [std_sep], '2'

echo 'Der Cursor befindet sich an Position ' # [pos]
echo 'Die Markierung ist ' # [len] # ' Zeichen lang.'

rem Pfad zu dem Bild eines IMAGE-Widgets ermitteln
getdialog [imagefile] = 'myDialog:Image1', 'IMAGE'
echo 'Die Bilddatei befindet sich hier: ' # [imagefile]
```

**Bemerkungen:**

Die Anweisung NEWDIALOG ermöglicht das Erzeugen von Widgets.



## Debugger Kontrollanweisungen

### DEBUGMODE

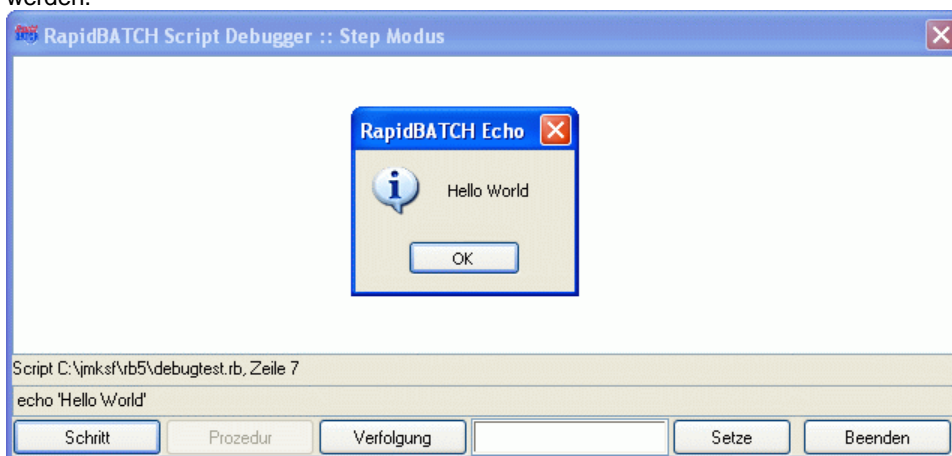
#### Syntax:

DEBUGMODE Modus

#### Beschreibung:

Setzt den RapidBATCH-Debugger auf einen von drei Debug-Modi. Mögliche Parameterwerte sind STEP für schrittweises Debugging, TRACE für Verfolgungsdebugging und NONE, um den Debugger auszuschalten.

Der Debugger kann somit vom Script selber mitten im Programm, z.B. an einer kritischen Stelle, gestartet oder beendet werden.



#### Beispiele:

```
echo 'Ein Debug-Test...'

rem Debugger starten
debugmode step

rem Weiter geht's im Debug Modus...
echo 'Hello World'
end
```

#### Bemerkungen:

Diese Anweisung ist nur im Interpreter der RapidBATCH Professional Edition enthalten. DEBUGMODE wird vom Compiler ignoriert und ist nicht in compilierten Programmen verfügbar.

Sie können mit der Umgebungsvariablen RB\_DEBUGMODE bereits vor dem Starten des Interpreters einen Debug-Modus festlegen. Mögliche Parameterwerte sind dabei dieselben wie bei der DEBUGMODE-Anweisung.

### DEBUGVARS

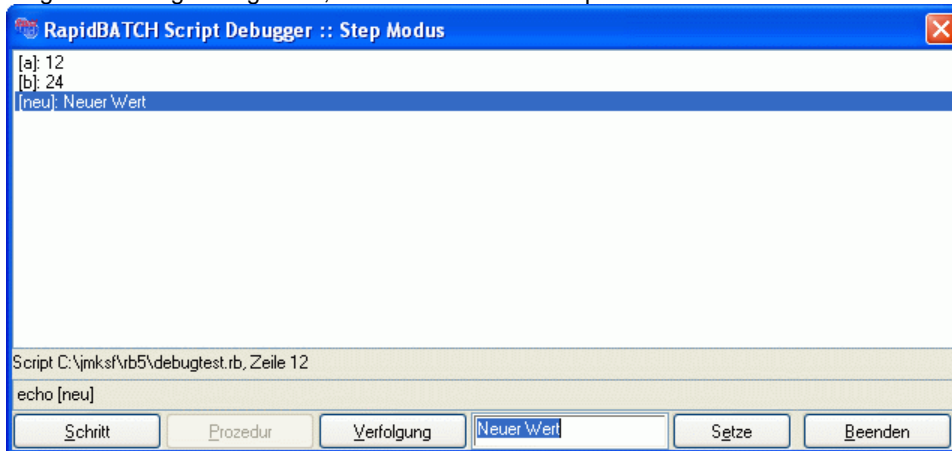
#### Syntax:

DEBUGVARS var1,var2,var3 ... varn  
DEBUGVARS Modus

#### Beschreibung:

Fügt die in einer Variablenliste angegebenen Variablen zum Debug-Dialog hinzu, um diese zu beobachten bzw. zu bearbeiten/modifizieren. Alle Variablen müssen durch Kommata (,) voneinander getrennt sein. Anstatt einer Variablenliste

sind auch die zwei Werte AUTO und NONE möglich. AUTO fügt jede neu deklarierte Variable automatisch zur Beobachtung dem Debug-Dialog hinzu, NONE schaltet diese Option wieder aus.



## Beispiele:

```
debugmode step
rem Nur [a] und [b] überwachen
debugvars [a], [b]
[a] = '12'
[b] = [a] * '2'
[c] = [a] * [b]

rem Automatisch alle neu deklarierten Variablen hinzufügen
debugvars auto

[neu] = 'Neuer Wert'
echo [neu]
```

## Bemerkungen:

Diese Anweisung ist nur im Interpreter der RapidBATCH Professional Edition enthalten. DEBUGVARS wird vom Compiler ignoriert und ist nicht in compilierten Programmen verfügbar.

Sie können mit der Umgebungsvariablen RB\_DEBUGVARS vor dem Starten des Interpreters bereits zu beobachtende Variablen festlegen. Möglich sind die selben Parameter/Werte wie bei DEBUGVARS.

## Vordeclarierte Variablen

### [active\_window]

**Beschreibung:**

Enthält den Fenstertitel-Text des aktuellen Vordergrundfensters.

Verwenden Sie [active\_window], um z.B. mit der SENDKEYS-Anweisung Tastaturbefehle an das aktive Fenster zu senden.

### [app\_icon]

**Beschreibung:**

Setzt das Programmicon für alle Dialoge.

In compilierten Scripts wird automatisch das hinzugelinkte (oder standardmäßige) Icon der EXE-Datei verwendet, in interpretierten Scripts das RapidBATCH-Icon des Interpreters. Über [app\_icon] kann dieses Icon nun auf eine externe .ICO-Datei, die sich z.B. im selben Verzeichnis wie das Script befindet, gesetzt werden. Dieses Icon erscheint dann in der linken oberen Ecke der Dialogfenster bzw. bei TRAYMENU als Traybar-Icon.

Wird [app\_icon] auf einen Leerstring gesetzt, so wird wieder das Standard-Anwendungsicon verwendet.

### [case\_sensitivity]

**Beschreibung:**

Hat die Variable [case\_sensitivity] den Wert -1 (bzw. [false]), so wird bei den Funktionen GETTOK, GETPOS, REPLACEVAR und CNTVAR nicht zwischen Groß-/Kleinschreibung unterschieden; Zum Beispiel werden, im Falle von REPLACEVAR, wenn der Suchstring "Hallo" ist, auch Substrings ersetzt, die "HALLO", "hallo" oder "HaLIO" lauten, wenn [case\_sensitivity] als -1 gesetzt wurde. Andernfalls nur der identische String "Hallo", mit großem "H" und kleinem "allo", wie er übergeben wurde.

### [clipboard]

**Beschreibung:**

Ermöglicht den Zugriff auf die Windows-Zwischenablage.

Wenn [clipboard] ein Wert zugewiesen wird, wird er automatisch in der Zwischenablage verfügbar und kann in anderen Anwendungen eingefügt werden.

Wenn [clipboard] ausgelesen wird, enthält sie den aktuellen Textwert, der in die Zwischenablage kopiert wurde (z.B. aus Word).

Wenn es sich bei dem in der Zwischenablage befindlichen Wert nicht um reinen Text handelt (z.B. Bilder, Dateien), so enthält [clipboard] einen Leerstring.

### [command]

**Beschreibung:**

Die Variable [command] enthält die Kommandozeilenparameter, die beim Starten des Scripts übergeben wurden. Bei interpretierten Scripts sind dies alle Werte, die hinter dem Script-Dateinamen beim Interpretierauf Ruf übergeben wurden, bei compilierten Scripts alle Werte nach dem Namen der EXE-Datei.

## [Echo\_Title], [Confirm\_Title]

### **Beschreibung:**

Definiert den Titel der Dialogfenster ECHO (Variable: [Echo\_Title]) und CONFIRM (Variable: [Confirm\_Title]). Standardwerte sind "RapidBATCH Echo" bzw. "RaidBATCH Confirm".

## [new\_line], [tab], [crlf], [pipe], [quot]

### **Beschreibung:**

Vielfach anwendbare Formatierungszeichen zur Text- und Dateiausgabe.

[new\_line] enthält einen einfachen Zeilenumbruch

[tab] enthält einen Tabulator-Vorschub

[crlf] enthält das Carriage Return und Line Feed Zeichen

[pipe] enthält das Pipe-Zeichen (|)

[quot] enthält ein einfaches Anführungszeichen (')

## [current]

### **Beschreibung:**

Enthält den Pfad des aktuellen Verzeichnisses.

Wechseln Sie das Verzeichnis mit der CHDIR-Anweisung.

## [curuser]

### **Beschreibung:**

Unter Windows NT, 2000, XP und 2003 der Benutzername des aktuell eingeloggten Benutzers. Bei Single-Benutzersystemen ("Family Logon") unter Win 95/98 und ME ist der Wert "unknown".

## [day], [month], [year]

### **Beschreibung:**

Das Systemdatum, aufgeteilt in drei Variablen, um verschiedene Datumsformate zu ermöglichen.

## [EditBox\_Ok], [EditBox\_Cancel], [EditBox\_Height], [EditBox\_Width], [EditBox\_X], [EditBox\_Y]

### **Beschreibung:**

Über die Variablen [EditBox\_Ok] und [EditBox\_Cancel] lässt sich die Beschriftung des OK- und CANCEL-Buttons im EDITBOX-Dialog ändern.

Mit den Variablen [EditBox\_Width] und [EditBox\_Height] lassen sich die Breite und Höhe des Dialogs in Pixel bestimmen.

Über [EditBox\_X] und [EditBox\_Y] lassen sich die X- und Y-Koordinaten in Pixel bestimmen, an denen die EDITBOX angezeigt werden soll. Standardwert von beiden Variablen ist 0, in diesem Fall wird der Dialog zentriert dargestellt.

## [eol]

### **Beschreibung:**

Die Variable [eol] (eol = end of line) wird von der WRITEFILE-Anweisung als Zeilenumbruchszeichen verwendet. Da WRITEFILE ja automatisch einen Zeilenumbruch hinter jede in eine Textdatei geschriebene Zeile hängt, lässt sich dieser über diese Variable definieren oder - je nach Anwendungsfall - gar komplett entfernen. Wird [eol] beispielsweise auf den Wert der Variablen [new\_line] gesetzt, werden Textdateien in einem für Unix-Programm lesbarem Format fortgeschrieben.

Standardwert für [eol] ist ein Windows-Zeilenumbruch, der als carriage-return + line-feed (Wagenrücklauf, Zeilenvorschub) definiert ist. Es kann jeder beliebige String als "Zeilenumbruch" definiert werden, der von WRITEFILE hinter jeden geschriebenen Wert angehängt wird.

## [errorcode]

### **Beschreibung:**

Die Variable [errorcode] ist eine von vielen Anweisungen und Funktionen benutzte Statusvariable, die Auskunft darüber gibt, ob eine Aktion erfolgreich war oder nicht oder welchen Status eine Aktion angenommen hat (z.B. welcher Button in einer MSGBOX-Meldung oder CONFIRM-Meldung gedrückt wurde).

In der Befehlsreferenz finden Sie im Bereich "Bemerkungen" jeweils Auskunft, welche Fehlerwerte [errorcode] von der jeweiligen Anweisung/Funktion zugewiesen werden können.

## [true], [false]

### **Beschreibung:**

[true] enthält den Wert 0, [false] enthält den Wert -1.

Anstatt diese, von vielen RapidBATCH Funktionen, Variablen und Anweisungen verwendeten Werte (0 = true, -1 = false) zu verwenden, kann (und sollte) man die Variablen [true] und [false] einsetzen.

## [InfoBox\_Enabled], [InfoBox\_Height], [InfoBox\_Width], [InfoBox\_X], [InfoBox\_Y]

### **Beschreibung:**

Über [InfoBox\_Enabled] lässt sich der Anzeigemodus des INFOBOX-Dialogs setzen, ob dieser eingeblendet (Wert 0 oder 'show') oder ausgeblendet (Wert -1 oder 'hide') werden soll.

Mit den Variablen [InfoBox\_Width] und [InfoBox\_Height] lassen sich die Breite und Höhe des Dialogs in Pixel bestimmen. Über [InfoBox\_X] und [InfoBox\_Y] lassen sich die X- und Y-Koordinaten in Pixel bestimmen, an denen die INFOBOX angezeigt werden soll. Standardwert von beiden Variablen ist 0, in diesem Fall wird der Dialog zentriert dargestellt.

## [InputBox\_Ok], [InputBox\_Cancel], [InputBox\_X], [InputBox\_Y]

### Beschreibung:

Über die Variablen [InputBox\_Ok] und [InputBox\_Cancel] lässt sich die Beschriftung des OK- und CANCEL-Buttons im INPUTBOX-Dialog ändern.

Über die Variablen [InputBox\_X] und [InputBox\_Y] lassen sich die X- und Y-Koordinaten in Pixel bestimmen, an denen die INPUTBOX angezeigt werden soll. Standardwert von beiden Variablen ist 0, in diesem Fall wird der Dialog zentriert dargestellt.

## [ListBox\_Ok], [ListBox\_Height], [ListBox\_Width], [ListBox\_MultiSel], [ListBox\_X], [ListBox\_Y]

### Beschreibung:

Über die Variablen [ListBox\_Ok] lässt sich die Beschriftung des OK-Buttons im LISTBOX-Dialog ändern.

Mit Setzen von [ListBox\_MultiSel] auf den Wert 0 lassen sich mehrere Einträge im LISTBOX-Dialog selektieren, diese werden dann als Liste zurückgegeben, wo jedes selektierte Element durch das in [std\_sep] definierte Trennzeichen separiert wird.

Steht [ListBox\_MultiSel] auf -1 ist nur eine einfache Selektion möglich.

Mit den Variablen [ListBox\_Width] und [ListBox\_Height] lassen sich die Breite und Höhe des Dialogs in Pixel bestimmen. Über [ListBox\_X] und [ListBox\_Y] lassen sich die X- und Y-Koordinaten in Pixel bestimmen, an denen die LISTBOX angezeigt werden soll. Standardwert von beiden Variablen ist 0, in diesem Fall wird der Dialog zentriert dargestellt.

## [XPStyleActive], [Metric\_Menu\_Height], [Metric\_Caption\_Height], [Metric\_Edge\_Height], [Metric\_Edge\_Width]

### Beschreibung:

Die Variable [XPStyleActive] hat den Wert 0 (bzw. [true]), wenn das verwendete Betriebssystem Windows XP oder Windows 2003 ist und der aktuelle Benutzer das Windows XP Theme eingeschaltet hat. Beim klassischen Windows-Theme sowie unter älteren Windows-Systemen wie 2000 oder 98 enthält die Variable den Wert -1 (bzw. [false]).

Die Variablen [Metric\_Menu\_Height], [Metric\_Caption\_Height], [Metric\_Edge\_Height] und [Metric\_Edge\_Width] enthalten Informationen über die Höhe der Menüleiste, die Höhe der Titelleiste sowie Höhe und Breite der Umrandung eines Fensters. Verwenden Sie diese Variablen, um Ihre selbst-erstellten Anwendungsfenster optimal an Windows XP Themes sowie Windows-Systeme mit vergrößerter Fensterdarstellung anzupassen.

## [mouse\_x], [mouse\_y]

### Beschreibung:

Geben die aktuellen X- und Y-Pixelpositionen des Mauszeigers zurück.

Durch zuweisen der jeweiligen Variable wird der Mauscursor auf die gewünschte Position auf der entsprechenden Achse gesetzt.

## [null]

### **Beschreibung:**

Die Pseudo-Variable [null] wird eingesetzt, um beispielsweise einen nicht benötigten Parameter, der an eine Prozedur übergeben wird, anzugeben oder um den Rückgabewert einer Funktion, der nicht benötigt wird (was manchmal der Fall sein kann) abzufangen. [null] ist in dem Sinne keine Variable. Werte, die an [null] übergeben werden, werden im Speicher sofort verworfen; Wird [null] ausgelesen, enthält sie einen Leerstring.

## [OpenBox\_DefFileExt], [SaveBox\_DefFileExt]

### **Beschreibung:**

Definiert für den jeweiligen Dialog eine Standard-Dateierweiterung, die automatisch an einen im OPENBOX- bzw. SAVEBOX-Dialog eingegebenen Dateinamen OHNE Dateiendung angehängt wird. Die Angabe eines Punktes ist hier nicht erlaubt, nur die reine Dateiendung selber, z.B. 'TXT' für Textdateien.

## [ownname]

### **Beschreibung:**

Dateiname des Scripts. Bei interpretierten Scripts der Dateiname der .RB-Datei, bei kompilierten Scripts der Name und Pfad der .EXE-Datei.

## [RB\_Version], [RB\_Pro]

### **Beschreibung:**

Enthält die Versionsnummer der RapidBATCH-Version, mit der das entsprechende Script ausgeführt wird (z.B. "5.0"). Dies ermöglicht es z.B., in interpretierten Scripts verschiedene Quellcode-Abschnitte zu verwenden. Die Variable [RB\_Pro] gibt Auskunft, ob es sich bei der verwendeten RapidBATCH-Version um eine Professional Edition (Wert: 0) oder Standard Edition (Wert: -1) handelt.

## [ScreenWidth], [ScreenHeight]

### **Beschreibung:**

Geben Auskunft über die aktuelle Auflösung.

[ScreenWidth] enthält die Anzahl der horizontal verfügbaren Bildschirmpixel, [ScreenHeight] enthält die Anzahl der vertikal verfügbaren Bildschirmpixel.

Mit diesen Werten können z.B. Dialoge wie EDITBOX oder LISTBOX auf den gesamten Bildschirm maximiert werden, oder aber auch individuelle Dialoge zentriert werden.

## [std\_sep]

### **Beschreibung:**

Die Variable [std\_sep] enthält den Standard-Listenseparator.

Dies ist ein Zeichen, welches bei Listen, die z.B. bei LISTBOX oder OPENBOX übergeben werden, als Trennzeichen für die einzelnen Elemente verwendet wird.

[std\_sep] enthält defaultmäßig ein Pipe-Zeichen (|). Es kann immer nur ein einzelnes Zeichen als Separator definiert werden, wenn ein String mit mehreren Zeichen an [std\_sep] übergeben wird, wird nur das erste Zeichen als Separator verwendet.

## [time]

### **Beschreibung:**

Die aktuelle Systemzeit im Format HH:MM:SS.

## [windir]

### **Beschreibung:**

Enthält den Pfad des Windows-Systemverzeichnisses.

## [winvers]

### **Beschreibung:**

Gibt Auskunft über die verwendete Windows-Version.

Mögliche Werte sind 95, 95A, 95B, 95C, 98, 98SE, ME, NT4, 2000, XP, 2003 oder unknown, wobei letztere nur bei neueren Windows-Versionen als 2003 eintritt.



## Index Sprachreferenz und vordefinierte Variablen

[active_window] .....	132	[ScreenHeight] .....	136	GETREGVAL .....	108
[app_icon] .....	132	[ScreenWidth] .....	136	GETTOK .....	85
[case_sensitivity] .....	132	[std_sep] .....	137	GOSUB .....	78
[clipboard] .....	132	[tab] .....	133	GOTO .....	77
[command] .....	132	[time] .....	137	HALT .....	79
[Confirm_Title] .....	133	[true] .....	134	IF .....	72
[crlf] .....	133	[windir] .....	137	INCLUDE .....	79
[current] .....	133	[winvers] .....	137	INCLUDE_ONCE .....	79
[curuser] .....	133	[XPStyleActive] .....	135	INFOBOX .....	96
[day] .....	133	[year] .....	133	INPUTBOX .....	91
[Echo_Title] .....	133	BREAK .....	74	LETDIALOG .....	117
[EditBox_Cancel] .....	133	CALL .....	99	LETENV .....	107
[EditBox_Height] .....	133	CHDIR .....	98	LETFILEATT .....	104
[EditBox_Ok] .....	133	CNTVAR .....	87	LETFILEDATE .....	105
[EditBox_Width] .....	133	CONFIRM .....	89	LETFILETIME .....	105
[EditBox_X] .....	133	CONT .....	74	LETREGVAL .....	109
[EditBox_Y] .....	133	COPYFILE .....	100	LISTBOX .....	93
[eol] .....	134	COPYVAR .....	86	LOCKWORKSTATION .....	114
[errorcode] .....	134	DEBUGMODE .....	130	MCI .....	107
[false] .....	134	DEBUGVARS .....	130	MKDIR .....	98
[InfoBox_Enabled] .....	134	DEC .....	72	MOUSEEVENT .....	113
[InfoBox_Height] .....	134	DELFILE .....	99	MSGBOX .....	90
[InfoBox_Width] .....	134	DELREGKEY .....	108	NEWDIALOG .....	116
[InfoBox_X] .....	134	DELREGVAL .....	111	NEWFILE .....	99
[InfoBox_Y] .....	134	DISKFREE .....	106	NEWREGKEY .....	108
[InputBox_Cancel] .....	135	DISKSIZE .....	106	OPEN .....	99
[InputBox_Ok] .....	135	DISKTYPE .....	106	OPENBOX .....	95
[InputBox_X] .....	135	ECHO .....	89	PROC .....	75
[InputBox_Y] .....	135	EDITBOX .....	92	PUTFILE .....	103
[ListBox_Height] .....	135	ELSE .....	72	PWDBOX .....	91
[ListBox_MultiSel] .....	135	ELSEIF .....	72	RANDVAR .....	88
[ListBox_Ok] .....	135	ENDFUNC .....	76	READFILE .....	101
[ListBox_Width] .....	135	ENDIF .....	72	REM .....	80
[ListBox_X] .....	135	ENDPROC .....	75	RENAMEFILE .....	100
[ListBox_Y] .....	135	EXT .....	81	REPEAT .....	74
[Metric_Caption_Height] .....	135	FILEEXISTS .....	101	REPLACEVAR .....	87
[Metric_Edge_Height] .....	135	FILESIZE .....	100	RESET .....	72
[Metric_Edge_Width] .....	135	FINDFILES .....	103	RESUME .....	78
[Metric_Menu_Height] .....	135	FOLDERBOX .....	94	RMDIR .....	98
[month] .....	133	FUNC .....	76	SAVEBOX .....	95
[mouse_x] .....	135	GETASC .....	84	SENDKEYS .....	111
[mouse_y] .....	135	GETCHARAT .....	85	SHELL .....	98
[new_line] .....	133	GETCHR .....	84	SHORTCUT .....	106
[null] .....	136	GETDIALOG .....	123	SHUTDOWN .....	115
[OpenBox_DefFileExt] .....	136	GETENV .....	107	SYSTEM .....	98
[ownname] .....	136	GETFILE .....	102	TRAYMENU .....	97
[pipe] .....	133	GETFILEATT .....	104	TRIMVAR .....	86
[quot] .....	133	GETFILEDATE .....	105	UNTIL .....	74
[RB_Pro] .....	136	GETFILETIME .....	105	UPVAR .....	86
[RB_Version] .....	136	GETLEN .....	84	WAIT .....	80
[SaveBox_DefFileExt] .....	136	GETPOS .....	84	WRITEFILE .....	101